Языки программирования

Disclamer

Может содержать ошибки, а также исправления, отличающиеся от того, как писал лектор. Помимо материалов содержатся цитаты преподавателя.

Приношу благодарности Лене Шамаевой за помощь с некоторыми материалами и Никите Бобко за правки.

Домрачева Д.

Лекции 1, 2

Вводная часть (немного истории)

- PL/I — не выжил, был неудобен
- 1968 Algol-68 — совсем другое дело, создан учеными.
  - Pascal, C пошли от него.
  - "Замечательнейший" ЯП, но через 10 лет его пересмотрели.
  - W-грамматики, может генерировать потенциально бесконечное множество контекстно-полных правил, символы генерировались особым образом, но механизм достаточно простой.

Всякая дополнительная инфа

Ортогональность ЯП: "Этот термин был введен в информатике для обозначения некой разновидности независимости или несвязанности. Два или более объекта ортогональны, если изменения, вносимые в один из них, не влияют на любой другой. В грамотно спроектированной системе программа базы данных будет ортогональной к интерфейсу пользователя: вы можете менять интерфейс пользователя без воздействия на базу данных и менять местами базы данных, не меняя интерфейса." "Программист-Прагматик. Путь от подмастерья к мастеру" Э. Хант, Д. Тома

Чистые вычисления без изменения состояния памяти. Изменение состояния — главная задача оператора. Главный оператор в ЯП (каком?) — оператор присваивания. Операторы управляют вычислениями

// Pascal
v := e; //выражение превратилось в оператор, вычисляется значение v

В C нет оператора присваивания, но есть оператор-выражения. Операции с побочными эффектами, присваивание — побочный эффект (а так же инкремент, декремент и т.п.). Получился ортогональный язык.

конец фрагмента

- Ада — разработка на деньги военных, 50% — непосредственно разработка, 50% — последующее сопровождение.
  - Необходимость в последующей доработке и развитии языка
  - Необходимость в сокращении расходов на сопровождение => выделили несколько основных ЯП, чтобы вести на них разработку. В итоге выработанным требованиям мог удовлетворить 1 ЯП, был объявлен конкурс 1980: ЯП Pascal, Algol-68 и PL/I. Конкурс выиграли проекты, основанные на Pascal. В финале выбрали "зеленый" язык, который назвали Ада.
  - 1983 выработали окончательный стандарт.

2 основынх способа дизайна ЯП:

1. Сундук (принцип осознанных технологических потребностей) — осознается, что нужно для ЯП, и это включается в ЯП. Ада использует этот принцип, требования исследовали на протяжении 3 лет. Функции без побочных эффектов — включены, процедуры сочли избыточными.
2. Чемодан — берется только то, без чего нельзя обойтись, принцип минимальонсти языковых конструкций. Есть проблемная область, ЯП конструируется только для этой области.

Таким образом, Ада достаточно сложный ЯП, невозможно ни расширение, ни сужение стандарта, некоторая избыточность. Многословный язык. Основная проблемная область Ады — противоракетная оборона.Фактически яп можно считать мертвым, хотя есть GNAT — GNU транслятор для Ады. Встроенная проверка на границы массива, которую можно отключить. В C таких проверок в runtime нет. Индексирование массива (еще такое в basic и Fortran): a(i) ПО должно быть "реального времени", т.е. надежным и с гарантированнм откликом.

C: если поля нет у структуры, то раньше (в компиляторе K. Ричи) генерировалось предупреждение, компилятор ставил нулевое смещение, теперь выдается ошибка.

p->name;

В Аде нужна была надежность, поэтому:

1. Runtime поддержка и статический контроль компилятором;
2. Работа в реальном времени;
3. Читабельность (из-за этого многословность, но не факт, что разраюотчики добились этим, чего хотели).

–

- Java, 1995: первоначальная идея — пересылка программ по сети, создание апплетов (потом от этого октазались), так что Write Once Run Anywhere (WORA). Принцип работает из-за наличия Java-машин, но они не все совместимы: JVM (virtual machine) + JRTE (runtime edition, для каждой системы может быть свой, от этого и несовместимость).
  - JWI — интерфейс

Таким образом, любая попытка создать единый ЯП ведет к неудаче, каждый хорош в своей отрасли. => Нужно определить само понятие ЯП.

Виды программирования

1. Игровое (учебное)
2. Научное
3. Индустриальное

Нам нужно индустриальное

> Почему интерпретатор C — неуловимый Джо? Потому что его никто не ловит.

(Разговор за бэкенд и фронтенд) Go: динамическая сборка мусора, нет классов (в этом отличия от C++), >Go — сильно улучшенный C

Парадигма программирования

Совокупность всех навыков, методов, инструментов и систем

3 парадигмы:

1. Процедурная (императивная) — от архитектуры Фон Неймана
   - Fortran

- Assembler
- C — императивный
- Появление ООП (классы, состоящие из экземпляров. Instance variables; class variables - статические данные). Объектно-императивные ЯП:
  - C#
  - C++
  - python

2. Функицональная
   - Рефал
   - Lisp
3. Логическая
   - Prolog

Мой научный руководитель, уважаемый человек, пожилой... Ну сидел, извините меня, старый пердун, медленно по клавишам стучал, да вот я сейчас так же

Лекции 3, 4

Примеры парадигм программирования (ПП)

Все парадигмы зародились примерно в 50-е.

1. Императивная (процедурная) ООПП (объектно-императивн). Сейчас доминирует. Состояние, методы (определяют поведение объекта), алгоритмы. Сильно опирается на состояние. Изобретатель парадигмы — Алан Кей (Alan Key). SamllTalk — 1975. Главное в ООП (по Алану Кею) — посылка сообщений и их динамическая обработка. Императивное программирование зашло в тупик? (70-е)
2. Функциональная
3. Логическа

Начнем с императивной ПП.

C

ОС написаны на C (иначе пришлось бы прогать на ассемблере, C заменил его). В 2000-е ассемблерные программы начали переписывать на C++ Страуструп: > C++ is better C

```
#include <stdio.h>
// контекст трансляций, информация для компиялтора, имена (набор неких имен, неопределенных в программе, но испо
#include <stdlib.h>
/** имя — идентификатор (с точки зрения императивных ЯП), таким образом здесь — набор стандартных идентификат
в компиляторе C НЕТ стандартного контекста!!! int, float и т.д. не идентификаторы, а ключевые слова. Так же во многи
передаем headers <> — стандартная билиотека. Наши headers — " ". Далее — обработка препроцессором. Явное импорти
слово extern — дурной стиль!!! Не употребляется в C (но вот в ассемблере можно), Эвы не понимаете C, если использует
*/
extern int i; // m1.c — extern из m2.c
/**

*/

double i; /** m2.c — здесь double => может быть ошибка, несоответствие размеров. Даже если соответствуют, то тоже п
*/
```

И недостаток, и достоинство в том, что C — язык ассемблера. Контроль межмодульных связей отсутствует.
Дальнейшее развитие для C++ 20 — обсудить понятие модуля, но тогда будет ли это C++?

```
#include <stdio.h>
#include <stdlib.h>
#define BUF_SIZE 1024;
/** пишется большими буквами, так как это константа препроцессора, нужно это выделить (препроцессорные имена — н
```

```
struct bar {
    struct foo {int i;}
    i;// область видимости типа структуры отсюда и ниже
} c;
int i;
struct foo c1;

/** struct time: time(...) */

char buffer[BUF_SIZE];

/** в первой версии языка C — если функция ничего не возвращала, то по умолчанию она возвращала int:
main(argc, char ** argv) {} */

int main(int argc, char ** argv) // или int main()
{
    int ch;
    int index = 0;
    // ТАК ПИСАТЬ НЕ СТОИТ!!!

        buffer[index++] = ch;
        /** может быть buffer override — уязвимость программы. Таким образом хакеры, зная код, могут перетирать па
        SQL-injection — вставка вредоносного SQL кода. Поэтому сейчас производится предварительная интерпретация
        /** В C есть функция, которую нельзя употреблять! — gets(char *buf), нет никакого предохранения. Есть функ
        */

    }
    while ((ch = getchar()) != EOF) {
        if (index == BUF_SIZE) {
            fprintf(stderr, "...");
            exit 1;
        }
        buffer[index++] = ch;
    }
}
```

Но это неполное решение, могут быть строки только до 1024.

```
#include <stdio.h>
#include <stdlib.h>
#define BUF_SIZE 1024;
char * buffer;
int size;
int main(int argc, char ** argv) {
    int ch;
    int index = 0;
    buffer = (char *) malloc(BUF_SIZE);
    size = BUF_SIZE;
    while ((ch = getchar()) != EOF) {
        if (index == size) {
            realloc(
```

```
            /** увеличивать в 2 раза, но проблема realloc — нужны непрерывные куски памяти, поэтому нужна проверка. Н
*/
// C runtime — стандартная библиотека C (нет такого, как в Java — JRT)
        }
        buffer[index++] = ch;
    }
    for (index; index >= 0; index--)
        printchar(buffer[index]);

    // будет работать очень медленно, в каждой итерации — системный вызов, а это прерывание, замедляет программу.
}
```

В C89 появился еще VLA — variable length array. Еще минусы C:

> В C совершенно уродский синтаксис.

И: > Если программа на C не будет эффективнее, чем на любом другом ЯП, то вы неправильно написали программу на C.

? для ассемблера ?

$$\frac{T_L}{T_{AS}} > 1$$

$$T_{AS} > T_L$$

Для C (L — любой ЯП):

$$\frac{T_C}{T_L} < 1$$

C#

Все на свете — класс.

```
public class Program{}
```

Есть методы экземпляра и методы класса.

```
public void Main(Sring[] args) // язык со строгой типизацией
{
    String s = System.Console.In.ReadToEnd(); // String — из стандартного контекста, string — ключевое слово
    /** до конца файла. Аллокация памяти, алгоритм — какой-то, неизвестно, как именно, универсальный алгоритм мож
    */
    /** C: можно модифицировать quicksort, вместо среднего элемента массива — медиана трех (в среднем будет хуже, н
    */
    for (int i = s.Length - 1; i >= 0; i--)
        System.Console.Write(s[i]); // такой код считается достаточно низкоуровневым
    /** Идея: просто указать, что делать с объектом, т.е. не указывается цикл, а указывается последовательность, в како
    char[]c = s.ToCharArray();
    c.Reverse();
    System.Console.Write(c);

    /** Проблема класса — он и модуль, и тип данных. Иногда удобно, а иногда и нет.
    System — пространство имен
    Console — имя класса из System
    In — статический член класса Console
    У класса TextReader есть метод ReadToEnd
```

```
    Получается дуализм
    */
    /** нигде нет подключения модулей: это можно сделать в опциях проекта*/
}
```

Python

Задумываался как просто “чемоданчик с инструментами”. Интерпретируемый язык. Для него работает REPL — Read Evaluate Print Loop. Интерпретатор берет какую-то часть (оператор) и выполняет его. В Питоне короткие программы (для простейшей нужо два оператора, один из которых import)

```
import sys

raw = sys.stdin.read()
l = [c for c in raw] # list comprehension
l.reverse() # ничего не возвращает (проверено в интерпретаторе), у лектора l.reverse сразу было в join
print(''.join(l)) # раньше в python2 был оператором, потом в python3 стал функцией

# a= ...

b = a[0:len(a)] # копирование всего массива
b = a[:-1] # копирование без послледнего элемента
# правильное "короткое" копирование:
b = a[:]
```

Можно коротко написать:

```
import sys
print(sys.stdin.read()[::-1]) #[::-1] — reverse
```

> В Lisp: (abc) воспримется как один символ, поэтому (print reverse read) выведет (abc). Надо подавать как (a b c).

C++

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator> //  содержит полезные итераторы (итераторы работы с вводом-выводом и класс inserter: для вст
// для вектора существует только back_inserter, в целях эффективности
using namespace std;
int main() {
    vector<char> v;
    copy(istream_iterator<char>(cin) /** аналог функции begin()*/, istream_iterator<char>(), back_iterator(v));
    copy(v.rbegin(), v.rend(), ostream_iterator<char>(cout));
    return 0;
}
```

Go

Есть интерфейсы и динамическая сборка мусора, нет ООП. Модульный язык. Стандартный контекст достаточно маленький, все, что нужно, должно быть импортировано.

```
package main
import ("fmt"; "io/ioutil"; "os"; "strings") //  модули
func main() {
    rdr := os.Stdin // автообъявление с помощью :=, тип выводится из типа правой части
    // в C++ появилось auto и decltype()
    b, err := ioutil.ReadAll(rdr) // вернется строка (последовательность байтов) и признак ошибки, если есть
    if err != nil {
        panic("bad input")
    }
    s := string(b)
    x := string.Split(s, "")
    for i := len(x) - 1; i >= 0; i-- {
        fmt.Print(x[i])
    }
}
```

Лекции 5, 6

Обобщенная парадигма программирования

Входит в императивную парадигму. Есть понятие состояния. Суть — статическая параметризация типов данных:

- надежно, так как типы контролируются во время трансляции,
- эффективно

В Питоне не может быть статической параметризации (там динамическая, так как на этапе трансляции типов еще нет, есть объекты и их взаимодействие).

Функциональная парадигма

Лисп

1959 Pure Lisp — образец функционального программирования. Из Лиспа выкинули все, связанное с процедурностью (так как иначе — мультипарадигмальность). List Processing Но на самом деле — язык обработки S-expression, символьное выражение (список — самый частый представитель). Самый популярный стандарт Лиспа — Common Lisp.

SET Q EXPR // Чистое понятие состояния

- Данные Крайне простые структуры данных, с котроым он работает. Атом — данные, не обладающие внутренней структурой. Символ — идентификатор + некоторое расширение (a, +, T. . .). Базовые лексемы — () . ; остальное — символы. Отсутствует понятие “строка”. S-выражение (точечная пара): a.b, где a и b — либо атомы, либо S-выражения. S-выражения — 2 ячейки, в одной указатель на первую переменную, во второй — на другую (либо на таблицу атомов, либо на другой список/S-выражение). CAR — head списка CDR — tail списка Специальный атом — пустой список, нулевой указатель, можно рассматривать как список (одновременно и атом, и список). Список — S-выражение особого вида, где последний элемент — пустой список (2 ячейки, первая — указатель на спиок/что-то там еще, вторая — пустой список). Фактически, список в Лиспе — это дерево. Левая часть может быть либо атомом, либо списком. Правая часть всегда ссылка на список.

a.NIL // короткое обозначение — (a)
a.(b.NIL) // указатель на a . указатель на (указатель на b . пустой список)

- Операции Вычисление атомов: первым символом должен стоять тот, с которым связано функциональное значение, а дальше:

(+, a1, ..., an)

Если нет побочных действий, то порядок вычисления неважен. Результат вычисления — то, что выдает функция, например, для +: список из одного элемента: (+ 1 2 3 4 5) -> (15) Должен быть набор встроенных функций:

(CAR S) // head of S, либо атом, либо список
(CDR S) // tail of S, всегда список (пустой или нет)
(read) // список из результата функции read
(1) // список из одного символа
(abc) // abc воспринимается как один символ
(a b c) // список из трех элементов
reverse(read) // если read вернет список, то будет reverse от списка, если атом, то атом
print(reverse(read)) // вывод списка в обратном порядке
(NULL S) // если S — пустой список, то выдаст T, если ложь, то пустой список (аналог False)
(COND (S1) res1
(S2) res2
...
(Sn) resn
) //вычисляется, пока не сработает одно из условий
(COND (S1) res1
(T) res2
) // аналог if, для простоты будем использовать (IF (S1) res1)
(append S1 S2) // объединение списков (сначала S1, потом идет S2 в конец). Можно записать через cons (?)
(cons a.b) // конструирование списка
(ATOM S) // атом или нет

Позиции рассмотрения языка:

- Технологическая: должен отражать те или иные технологические потребности в зависимости от технологической ниши языка (некоторой отрасли яп), например, системное программирование (здесь лидирует C, вытеснив ассемблер), научно-технические вычисления (лидирует/-овал Fortran, теперь вытесняется R, matlab).
- Авторская добавление какой-либо функциональности (модули, динамическое программирование) при необходимости.
- Реализаторская
- Семиотическая
- Социальная

RAD — rapid application development REPL — read-evaluate-print

Функция reverse на Lisp

(defun Sym (Sym1 Sym2 ... Symn) S) //Symj — аргументы
(defun reverse1(S)
(IF (NULL S) ()
(append (reverse(CDR S)) (CAR S)) // тут все упадет на CAR, если там атом, так как аргументы append — 2 списка, п
(append (reverse(CDR S)) (cons (CAR S)))
)
) // это все будет долго и неэффективно работать, для reverse должно быть достаточно одного прохода (с вспомогательно
(defun shift(l r) // берет голову l и переносит в r
(IF (NULL l) r
(shift (CDR l) (cons (CAR l) r))
)

```
)
(defun reverse1(S)
    (shift S ())
)
```

Логическая парадигма

Prolog

Каноническая форма Хора:

$$P^1(X) - {P_1}^1(X), {P_2}^1(X), ..., {P_n}^1(X)$$
$$P^2(X) - {P_1}^2(X), {P_2}^2(X), ..., {P_n}^2(X)$$

Работает как конъюнкция. Это — набор предикатов, которые описывают предметную область. Также туда входят факты (без переменных выражения). Факты:

```
MAN(SOKRAT)
MORTAL(X):-MAN(X)
A:-B // A верно, если верно B (если B, то A, читаем справа налево)
```

Запрос:

```
?:-MORTAL(SOKRAT)
```

Для доказательства система сама пребирает все варианты, отождествляет каждый вариант с X. Или доказывает, что запрос лодный, или находит X, при котором запрос верен. У Prolog 2 семантики:

- Декларативная
- Процедурная (вычислительная)

Есть понятие списка (и пустого тоже), сопоставление списка с образцом слева направо по длине списка. Предикат — функция, которая дает True или False.

```
APPEND(X, Y, Z) // истина, если z - результат конкатенации x и y
APPEND([1, 2], [3], X) // будет искать X для истины
APPEND([1, 2], X, [1, 2, 3]) // будет искать X
APPEND([1, 2], [3 | X], [1, 2, 3]) // X - пустой список
APPEND([1, 2], [X | Y], [1, 2, 3]) //  много вариантов
```

Отождествление идет в глубину.

```
REVERSE1([], []) // факт
REVERSE1([], X, Y)
REVERSE1([X | Y], Z) :- APPEND(W, X, Z), REVERSE1(Y, W)
// X - самая короткая непустая голова (ее сначала
// смотрят, потом более длинные части)

SHIFT(L, R, Z)
SHIFT([], X, X)
SHIFT([H | T], R, Z) :- APPEND([H], R, W), SHIFT(T, W, Z)

// можно задавать запросы:
```

?:- REVERSE1(X, [1, 2, 3])
?:- REVERSE1([1, 2, 3], X)

[P(X)] PROG [Q(X)]
// Циклы:
[P(X)] while (B) S [mot B(X)]
// Предикаты инварианта цикла INV(X) не мняется внутри цикла!

[INV(X)] while (B) do S [INV(X) S [INV(X)] S [INV(X)]
// или
[INV(X)] while (B) do S [not B(X) loop INV(X)]

Лекции 7, 8

REFAL

60-е годы, придуман физиком Турчиным. Нужен был формализм для обработки сивмольной информации. Во всех смыслах “Марковский” язык (алгоритмы Маркова). В языке: либо символ, либо выражение (с номерами, чтобы отличать их).

s.n // символ
s.1
e.m // выражение
e.2
s.1 e.1 s.1 // образец. такому образцу соответствует строка, которая начинается и заканчивается на один и тот же символ. сп
поле ввода = преобразование

```
$ENTRY GO
{
    = <Prout  < REVERSE <card>>>; // пустое поле ввода
}
REVERSE
{
   // не пустая строка => символ и произвольная строка (мб пустая):
   s.1 e.1 = < REVERSE e.1> s.1 /; // / — завершение
   // если пустая строка:
    = ;
}
```

Базисные понятия ЯП

(Для всех парадигм)

1. Данные
2. Операции
3. Связывание

Данные и операции

Программы обрабатывают данные. В каждом ЯП есть понятия данных и операций (самые базисные понятия, для общности ЯП их сложно уточнить, так как суть весьма прозрачна и понятна). Для каждого конкретного ЯП эти понятия можно определить очень точно.

Немного о C: Простой в плане понятий данных и операций, но стандарт очень сложен, не все реализации,

например, гарантируют “2+2=4” (представим двухбитный процессор: 2+2 вызовет переполнение). Не гарантируется правильность вычисления:

l(i++, ++i)

будет зависеть от компилятора (порядок вычисления).

p != NULL && *p > 0

— ленивые вычисления, пример зависимости от порядка вычислений.

В C:

strlen(s) // операция

В C++:

// basic_string<char>
string.size() // выдаст данные

То, что является данными, может являться результатом выполнения каких-то операций. Ленивые вычисления (lazy evaluation) — вычисление (взятие данных из структуры и тд) происходит ровно в тот момент, когда это нужно. Например, в python2:

range(N) # список от 0 до N - 1 [0; N)
xrange(N) # генератор (сущность)

В python3 уже нет xrange, он перешел в range. Список можно получить как list(range()).

C#:

```
class X {

    public int Y;
}
/**/
X a = new X();
a.Y = 1;

// "смотрите: настоящий ООП"
class X {
    private int _y;
    public int Y { // публичное свойство, нужны get и set
        set { _y = value; } // value - неявный параметр set
        get { return _y; }
    }
}
/*
*/
X a = new X();
a.Y = 1; // вызовется set с value=1, это не просто присваивание.
// наглядно реализовано в интерфейсах, например
/*
window w; // нужно привязать окно к какой-то точке
w.x = 0; // на самом деле изменение геометрии не гарантируется, владелец окна может это запретить. значение меня
w.y = 1;
*/
int j = a.Y; // вызовется get
```

Некоторый дуализм данных и операций, но с точки зрения теории операции первичны. ? types are not values ?

Абстрактные типы данных — АТД (ADT)

Данные харакатеризуют:

- Диапазон значений
- Операции над ними (даже в большей степени!)

АТД характеризуют ОПЕРАЦИИ! (происходит отход от важности значений данных).

В чистых ОО ЯП АТД превратились в интерфейсы — "голые" сигнатуры операций (допускаются статические данные). Когда придумывались АТД, имелось в виду, что описываться будут не только сигнатуры, но и свойства этих данных. Например: определим стек и операции в нем — pop(s, x) и push(s)

$$\forall stack(s) \forall value(x) push(s, x) => pop(s) = x$$

Есть типы данных (ТД). Есть объекты данных (над чем выполняются операции) Есть значения, каждое принадлежит какому-то ТП. Понятие "переменная" есть в любом ЯП. Отличие значения и переменной — (есть из переменной можно взять значение и можно его изменить — это императивная парадигма) у переменной есть имя, а значение, как правило, анонимно, например "3" — целый тип данных, но без имени. Можно дать имя значению, тогда это будет константа — частный случай переменной. Нотации не имеют понятия переменных: LAMBDA(X) (+ 1 X) — лямбда-функция, ее можно вызвать, вернуть. А если LAMBDA(X) (+ X Y) — Y переменная из области видимости:

```
// Lisp
(defun ADDX (Y)
   (LAMBDA (X) (+ X Y)) // Y
)

// JavaScript
function addY(Y) {
   return X => X + Y
}
var f = addY(5) // f - функция, 5 оказалась захвачена в функции только на чтение, функциональное значение безымянно
f(2) // 7
// переменные с типом данных не связаны
```

В C# между типами данных есть отношение наследования, переменная принадлежит какому-то ТД, есть статический тип, который не изменяется, и есть динамический тип — то, что туда положили на самом деле. ССылка на значение — X a = new X() — a становится ссылкой на значение, объектно-референциальный ЯП (C#, Java...). a может потом пменяться: a = new Y(), new X() останется того же типа, так как анонимное значение, и потеряется, если не сохранить на него ссылку. Массивы тоже являются референциальными.

Область видимости (scope)

Связана с понятием имени В том блоке, где объявлена. Или внутри класса (наследуемые классы образуют вложенные области видимости). Все ЯП являются проективными (одна структура вкладывается в другую). С именем связано понятие определяющего вхождения (декларация, определение). Видимость тэгов структур в C — от определяющего вхождения до конца файла (даже если вложено).

```
struct foo
{
   int i;
   struct bar
```

```
    {
        int i;
        int foo;
    }
} foo, bar; // В C можно (для тэгов отдельная таблица), в C# и Java так нельзя
```

Нужна была совместимость с UNIX, а там есть struct time time.

Область видимости и несколько определяющих вхождений? Раньше могло быть только одно для каждого имени. Появление перегрузки для имен функций (верно почти для всех ЯП, где есть понятие перегрузки). Понятие определяющего вхождения и использующего вхождения. В некоторых ЯП можно использовать переменные без декларации, как тогда определить, что это то самое определяющее вхождение?

```
function f(i) {
    if (i < 1) {
        x = 1 // что такое x?
    } // если транслятор увидит x, то он его перенесет выше как undefined, а потом задаст значение в ветке if. Но что
    else {...} // в некоторых расширениях JavaScript переменные обязаны быть определены (JSX)
}
```

Область действия (extent)

Связано с понятием значения (но не имени). Являются проективными во всех ЯП (вложенность друг в друга). В функциональных ЯП:

```
(defun f(x) (
    // тут x имеет одно значение
    )
) ( LET x () ) // переменная попала в захват и продолжит действовать (действует значение)
```

В императивных языках область действия распространяется и на переменные.

```
function counter() {
    var n = 0
    return () => n++
}// вернет функциональное значение без параметров и захватит n, поэтому у n  будет область действия как и у возв


...
counter()() // вернет 0
counter()() // тоже вернет 0, так как разные значения n

alert(counter()())// (вызов pop-up), вернет 0
alert(counter()())

var foo = counter()

alert(foo()) // 0
alert(foo()) // 1 - захвачено значение
```

Можно захватывать только read-only значения. В Java захват лямбдой переменной из внешней области разрешен только для final или effectively final переменных (констант или переменных, для которых компилятор может определить, что они являются константами) closure X => X + Y // захват переменной

> В C++ throw, а не raise, потому что C++ писался под UNIX (должна была быть совместимость), а там raise — посылка сигнала самому себе.

Все объекты данных имеют набор атрибутов. Например, в императивных ЯП — адрес

```
&v // C
v'ADDR // Ada
```

Можно передавать по значению и по ссылке В функциональных ЯП нет такого понятия. Значение как-то передается (зависит от реализации), например, в Лиспе, атом — ссылка на таблицу атомов, передастся эта ссылка. “Как хранятся, так и передаются” В императивных ЯП нужно четко понимать, ссылка это, или значение.

```
class X {
    public static void foo() {
        int i; // будет храниться в виде 4 байтов
        Integer j = 1; // объект, класс-оболочка, в него завернуто значение, j — ссылка на анонимный объект в динамическ
        int a[], b[]; // появилась ссылка
        a = new int[N]; // разместили массив в памяти
        b = a; // указывают на одно и то же
    }
}
```

Связывание (binding)

- Статическое связывание (до точки входа в программу)

static int i = 0; //(может осуществляться компоновщиком или загрузчиком)

- Квазистатическое связывание Связывание локальных переменных с адресом (похоже на статическое, но формально происходит во время выполнения программы, вычисляется статическое смещение относительно регистра стека — между ESP и EBP)

```
void f(int i) {
    int j;
}
// в ассемблерном коде: (EBP - указатель конца фрейма на стеке)
asm {
    MOV EAX, j[EBP]
};

/* */
static int q = 0;
asm {
    MOV EBX, q
}
```

Все становится плохо, когда доходит до VLA (variable length array) в C99. Нужно к адресу a прибавлять длину a в байтах, и после нее загружать b. Мы не можем знать статическое смещение всех переменных в эом случае!

```
char a[i], b[i];
/**/
asm{
    MOV EAX, b[i[EBP]] // ???
}

// FORTRAN
SUBROUTINE P(N)
```

DIMENSION REAL A(N)

C#:

```
void foo(int i)
{
    int[] a = new int[N];
    a[i] = 0; // квазистатическая проверка 0<=i<N, как если бы компилятор знал значение i
}
```

C++ STL: “Бескомпромиссное стремление к эффективности”: индексирование не проверяется, это не дает прибавку к эффективности. Проверяется только vec.at(i) (квазистатически).

- Динамическое связывание Атрибут связывается со своим значением в момент выполнения. Динамическое связывание адреса и динамических переменных.

Лекции 9, 10

Связывание

Конструкция ЯП связывается с набором атрибутов (выбор из некоторого конечного множества)

- Переменная и адрес
- Момент связывания: квазистатическое связывание переменной с адресом (статическое смещение относительно указателя на фрейм). Еще может быть статическим и динамическим.
- Время жизни связывания. Для императивных: совпадает с временем жизни переменных. Не для всех языков это так: чем ниже уровень языка, тем это вернее (связывание переменных с адресом характерно для низкоуровневых ЯП).

```
// в ЯП со сборщиком мусора может возникать дефрагментация памяти
byte[] ab = new byte[N]; // динамический массив, ссылка в C#, подверженная сборщику мусора и менеджеру памяти
unsafe // помечается атрибутом unsafe
{
   fixed(byte *p = ab) // происхоит фиксация адреса в p
   {
      p++; // внутри фиксированного блока можно использовать malloc и т.д., и таким образом можно делать по сути "
   }
}
```

- Переменная и значение. Связывание на уровне языка.
  - Императивные языки. Переменная отличается от константы временем жизни связывания. В какой момент происходит связывание константы? Ограничение накладывается на время жизни связывания, и it depends. К. Ричи был против констант в C++, так как не понятно, в какой момент вычислять константы. В C вычислялись на этапе препроцессирования или компиляции, а константы типа “const X a;” — ?

```
#define H (B - A)/N // неправильно
#define H (B - A)/(N) // верно, так как вместо N тоже может быть макрос

const int N = 10;
int a[N]; // — код из C++, но не C.
```

    В C++ появились прототипы функций, потому что компилятор генерировал код, основываясь на том, передавать ли параметр по ссылке или по значению. В C прототип функции изначально был просто “type function()”, компилятор даже не проверял параметры.

- –//–
  - В функциональных языках остается связывание вызова фактических переменных функций и формальных.

  - И там и там связывание динамическое, в функциональных языках время жизни связывания совпадает с временем жизни переменной. Связывание происходит раз и на всегда, нет понятия состояния, переменная не меняет значение. Переменные ведут себя как константы в императивных ЯП.

В C# конкретизация шаблонов происходит во время трансляции.

В C#:

```
const int N = 10; // const по умолчанию статические, связывание выполняется статически
int M = 10; // должно быть членом какого-то класса, если не статический объект, то инициализация происходит во
readonly int L = 10; // динамическая инициализация, время жизни связывания совпадает с временем жизни перемен
// инициализация статических объектов (эффективно) реализуется в отдельном статическом потоке, запускаемые си
```

В C++:

```
const X a; // инициализация происходит в runtime
```

- Переменная и тип данных. > Интерпретатор языка C никому не нужен (хотя его можно написать), так как он будет работать в десятки раз медленне, чем откомпилированный код.

Компилируемые языки:

$$\frac{T_{comp}}{T_{inter}} \gg 1$$

Типы данных и операции

Базисные типы

- Скалярные типы компилятор знает про эти типы. Не имеют внутренней структуры.
- Структурные типы

  Компилятор C “ничего не знает” о типах из стандартных библиотек В других языках есть мимикрирующие под стандартную библиотеку

Скалярные базисные типы данных в ООЯП

- Числовые типы данных
  - Целочисленные
  - Вещественные
- Логический тип данных
  - В C не было
- Символьные типы данных
- Перечислимые типы данных
  - До 2005 не было в Java
  - Создатель C#: это красиво и удобно при использовании IDE
- Диапазоны
  - Впоследствии вылетел из ЯП. Раньше использовался для задания массивов: “‘Pascal array[0..N-1] of T; // диапазон, но задан неявно

```Modula-2
array.IndexType of T; // квадратные скобки — свойство диапазона, а не массива, граница изменения индексов
Type T = [1..N];
Type T1 = CARDINAL[0..N];
```

- Указатели/ссылки
  - В некторых языках нет указателей, а ссылка не является отдельным типом (есть референциальный тип данных — классы, объекты, массивы, интерфейсы)
- Функциональные типы данных
- Строки (могут быть отнесены и к структурным).
  - В python и Go нет символьного типа данных
  - Как правило, неизменяемый объект
  - В C++ — тип из отдельной библиотеки

  Оберон — ООЯП, в отличие от Modula-2, но при этом более простой и компактный. Компилятор на Обероне занимает около 4к строк. С выходом Оберон-2 компилятор стал занимать 400-500 строк.

ARRAY N OF T // диапазоны выпали из Оберона
ARRAY OF T // открытый массив, динамический

Нет типов currency и date, хотя они были бы очень полезны. Существуют в отдельных библиотеках.

Есть типы, а есть оболочки — типы-заглушки (например, int и Int32 в C#, int и Integer в Java). Сделано, чтобы базисные типы стали объектами: boxing (упаковка типов).

Числовые типы данных

1. Представление (диапазон). Есть понятие регистров общего назначения с определенной разрядностью. int — базисный тип, родной дла заданной архитектуры (4 или 8), могут быть типы меньшей размерности: byte, short; int. long, long long — если не хватает int. В C# есть byte — 1, short — 2, int — 4, long — 8 (оболочки — UInt8, Int16, Int32, Int64). Аналогично в Java (фиксированные размеры, не как в C)

```
for (unsigned int i = 0; i < c.size(); i++) {} // size() — вернет size_t (unsigned int)
for (unsigned int i = c.size(); i >=0; i--) {} // будет зацикливание, смешение знаковых и беззнаковых операций
```

   Зачем нужны беззнаковые типы? — Нет беззнакового типа — нет беззнаковых проблем.

Операции сдвига для знаковых и беззнаковых типов работает по-разному (в знаковых: если знак — 0, то в начале забьются нули, если 1, то 1. Арифметический — деление на степени двойки, — и логический сдвиг — беззнаковый тип). По-хорошему, надо приводить к unsigned и делать сдвиг (логический сдвиг — операция »>). C++ изначально: смешивать знаковые и беззнаковые типы нельзя. Были перекомпилированы утилиты UNIX, ни одна из них не прошла проверку на несмешивание знаковых и беззнаковых типов. В итоге было разрешено смешивать. В C#: sbyte (знаковый byte, так как сам byte беззнаковый), ushort и т.д. Разрешены преобразования byte -> ushort -> uint -> ulong, byte -> short, ushort -> int и т.д.

В Обероне нет беззнаковых типов в целях минимизации.

Лекция 11, 12

Числовые типы данных

Числовые типы данных — целочисленные:

1. Беззнаковые?

```
i < c.size() // i - int, size() - uint
// integer overflow
checked {...}
unchecked {...}
```

   С точки зрения вычислений знаковый тип главнее. С точки зрения кода битовое представление знаковых и беззнаковых типов неразличимы (первый бит может быть после интерпретирован по-разному). Компилируемость определяется врменем связывания (если короткое, то проще компилировать, иначе — интерпретировать).

2. Представление (размерность)
   - Динамическая типизация (интерпретируемость). JavaScript (нет знаковых и беззнаковых, есть один тип — number, из-за этого все вычисления неточные). Python (int; long — вычисления с произвольной длиной, в Python3 отказались от двух типов, остался int, который по факту long из Python2 — по сути ссылка на объект из динамической памяти, строка = числу). Ссылки можно присвоить одна другой и они будут ссылаться на один и тот же объект (это проверятся в Python операцией is), и через ссылку можно добраться до свойств объекта.
   - Статическая типизация (компилируемость) — более производительны. Есть беззнаковые (кроме Java, там byte, short, int, long — все знаковые). C, C++, C#, Go (int8, int16, int32, int64 и byte, uin16, uint32, uint64; int, uint, uintptr), Swift (Int8, int16, Int32, Int64 и Uint8, Uint16, Uint32, Uint64; int, uint, uintptr). В основном объектно-референциальные языки. Проблема: что является родным размером int? Регистры общего назначения, если int будет меньше, то будет падение эффективности, не все возможности процессора будут использованы.

   Разговор за процессоры

	short	int	long	pointer	long long
LP32	2	2	4	4	-
ILP32	2	4	4	4	8
ILP64*	2	8	8	8	?
LP64	2	4	8	8	8
LLP64	2	4	4	8	8

IA-32 (1987) IA-64 (2001) -> x86-64

Вещественные числа

Можно представлять с плавающей точой, а можно с фиксированной (как в раннних компьютерах: числа хранились в десятичном формате по цифрам, каждая занимала 4 бита. В Python есть тип Decimal, который хранится подобным образом).

$$mantissa \cdot base^p$$

$$\frac{1}{base} <= mantissa < 1$$

В Ada (последний ЯП с фиксированными типами):

```
type T is digits 8; // точность 8 знаков
type MESH is delta H range L..R; // в пределах от L до R с шагом H, статические константы
```

По стандарту IEEE-754 (1985): float, double, long long double.

1. Стандартное представление 32 и 64 бита: 1 бит знак, мантисса (23 или 52 бита) и порядок — p' (8 или 11 бит). p' из всех 0 или 1 зарезервированы. p' = p + 2^(n-1) - 1 p' = p + 127 (float) p' = p + 1023 (double)
2. Underflow (не хватает мантиссы), возникает понятие Nan. Nan всегда не равен другому числу d (даже если d=Nan). В операциях с другими числами даст Nan. Нужно обрабатывать случаи возникновения Nan.
   - Overflow* (не хватает порядка), возникает понятие +-inf. 1/(+-inf).

   — Как поймать 6 львов? — Поймать 10 и 4 выпустить.

Логический тип данных

Логические операции так или иначе есть во всех ЯП, но типа данных может и не быть 0 = false #0 => true => 1 (в VisualBasic -1) and or not (xor — иногда, equal) f(e1, e2, . . . , eN): может вычисляться компилятором в любом порядке, например:

```
f(i++, ++i) // здесь порядок важен, но компилятор может вычислить в любом порядке
cout << i++ << ++i;
/**
((cout << i++) << ++i);
operator<<( operator<<(cout, i++), ++i) как правило сначала вычислится ++i, а потом i++ (так в большинстве реа
нестандартизовано
нежелательно употреблять выражения, вычисления которых зависят от порядка операндов.
*/
```

В процедурных языках для двуместных логических операций характерно ленивое вычисления:

```
i:=1;
while (i <= N) and (A[i] <> x) do
    i:= i + 1;
/**
не удовлетворяет стандартам, компилятор может вычислять в любом порядке. корректно работает только когда x ес
тогда нужно вынести второе условие
TurboPascal, C, C++.. требуют ленивые вычисления, все выражение не будет вычислено, если i за пределами массив
while (i < N && a[i] != x) i++;
*/
```

```
A x = null;
if (x != null) {}
```

В JavaScript и Python можно использовать логические операции с не логическими типами. В Python false — 0, None, [], {}, () (пустые структуры), в JavaScript — 0, null, undefined, [].

```
v1 && v2 // если v1 false, то v1. Если v1 true, то значение v2
v1 || v2 // если v1 true, то v1. Если v1 false, то значение v2
// верно для любого типа данных
```

Символьные типы

В Go, C нет символьного типа данных. В C ‘a’ — константа типа int, а в C++ — типа char. C: int getchar(). В какой-то степени char, wchar_t — символьные типы данных, но с арифметическими свойствами. C++:

```
basic_string<T>
string s; // basic_string<char>
wstring ws; // basic_sring<wchar_t>
```

В Python, JavaScript, Go — нет char, но есть string (на уровне базового понятия языка).

```
l=['a', 'b', 'c']
s = "".join(l)
```

Swift: пришел на замену Objective C.

Счетчик ссылок (в Objective C так строилось взаимодействие между объектами):

```
if (_ref_count == 0) {
    delete this;
    return 0;
}
```

> Яблоки не люблю вообще, особенно грызенные.

Лекции 13, 14

Символьные типы данных и строки

. ? ! ASCII 7 * JS, Python, PHP, ... — динамически типизируемые языки (не содержат типа char, только строки) * Статически типизируемые — как правило есть тип char, но в Go — тип rune (=uint32), нет отдельного символьного типа. Java: char => ushort.

70—80-е: какие представления текста использовать? * Однобайтовые кодировки символов (SBCS — single byte) * Двухбайтовые кодировки (DBCS — double byte) * MBCS

Система кодировки символов появилась в 19 веке (с появлением телеграфа), появились символы BackSpace (сдвиг каретки назад), Del (игнорировать послежний символ), CR (carriage return), LF (line feed), FF (form feed).

Однобайтовая кодировка: первые 128 символов — англ (американский стандартный код, ASCII7, ANSI-7), остальное можно использовать 128-255 под другие символы (национальные кодировки, а их достаточно много => как разграничить?). Появилось разграничение на алфавиты восточной и западной Европы. Была разработана ISO-Latin1 для западной Европы, для остальных — ISO-Latin2, ISO-Latin3... . Японцам и китайцам пришлось упростить иероглифы и придумать свою кодировку (сначала двухбайтовую, потом многобайтовую). Есть глобализованные приложения, а есть локализованные (все, что является текстом, должно отображаться в целочисленные идентификаторы, которые потом отображается в конкретные строки, загружаемые по их идентификаторам, строковые литералы, которые зависят от национального представления, в программе появляться не должны) => для подготовки локализации достаточно просто перевести текст. Но интерфейс должен адаптироваться к размерам ресурсов (например, размеру экрана). Глобализация — работа приложения везде, то есть нужно универсальное представление символов. Локализация требует перекомпиляции, но глобализация должна быть универсальной, требуется универсальное представление текстов => Появление UNICODE (1991) CP1252 CP1251 CP866 — кодировка кириллицы в MSDOS UNICODE не стандарт кодировки, это стандарт алфавита.

1. Стандартизуются алфавиты (наборы символов, Universal Character Set) — стандартизуются названия символов (например, small latin latter ‘a’).
2. Не стандартизуется представление (глиф). Например, в арабском: 23 буквы, у каждой до 4 написаний (для разных мест в слове).
3. Стандартизованы коды (уникальный номер символа) — Code Point. Кодировка отличается от Code Point тем, что говорит, как именно представить код в компьютере. UTF32 (то, как UCS переводится в кодировку 0—0x10FFFF). Для символов : можно либо выделить отдельные символы для букв с accents (é), либо сделать accent модифицирущим символом (если accent идет за буквой, то глиф будет объединением двух символов).

Наиболее используемые кодировки:

- UTF-8
- UTF-16

- UTF-32

Старый UNICODE получил название BMP (Basic Multilingual Plane). Символы стали обозначаться U+AB — это символ из BMP. Всего плоскостей 16. U+81FAB. 1..F. Максимальный символ — 0x10FFFF. > Все эти эмотиконс.. Эмоджики...

C: Все стандартные функции для работы со строками находятся в библиотеке <string.h>

```
strlen(char *);
wcstrlen(wchar_t *);
strcat(x, y);
wcstrcat(wx, wy);
```

BOM (byte order mark)

0xFFFE

- В машинах с прямой адресацией: FF FE
- Little endian: FE FF

В сети прямой порядок, поэтому нужны функции htons(), htonl(), wtohs(), wtohl() (при необходимости переводят формат). В файлах сначала указывается BOM (кодируется тремя байтами), поэтому пустые файлы не совсем пустые (хотя для UTF-8 BOM не имеет значения). Java, C#: char — 2 байта > UTF-16 — наихудшая из всех возможных кодировок UNICODE

JavaScript: line[0] — не всегда первый символ строки. Строку в символы можно превратить с помощью функции s.split("").

Go: Строки — последовательность байтов (не последовательность символов или рун). Последовательность байтов кодирует последовательность из UTF-8.

```
s string = "line"
fmt.printf(s[0]) // выдаст код символа l
s1 string = " "
frm.printf(s1[0]) // выдаст 32 — код пробела
len(s) // число байтов в кодировке
```

Тип rune, модуль utf8:

```
DecodeRuneInString(s) // выдаст пару (r rune, s size)
for i := 0; i < len(s);  // ++i — нет такой операции
{
    // автоматическая декларация
    r, sz := DecodeRuneInString(s[i:]) // вырезка, строка, начинающаяся с i-го байта
    // может быть только 110, 11110, 1110, 0_... (10 - продолжение)
    i += sz
}

// или
for r, sz in s {
    do_something()
    // ?
}
```

Длина “café” — 4, "cafe$\check{3}$01" — 5. Но отрисовывается 4 глифа.