Структурные базисные типы - кортежи

Вспомним про записи и массивы

Массив - последовательность *однотипных* переменных

Записи - последовательность *разнотипных* переменных

Массив - доступ по индексу - a[i] (за счет однородности и непрерывности) - одинаково работает для любого i

Запись - доступ по имени - rec.name - name известно во время трансляции (статически) => одинаково работает для любого имени поля внутри записи

Кортеж - структура данных, которая выглядит с точки зрения доступа как массив, но с точки зрения изменяемости (фикс. размер) - как запись.

Структурные базисные типы - кортежи

Пример 1 - Питон - полноправный тип данных

```
l = (1, "string", 0.3)
l[0] = -1 # - ERROR - read-only!!! => м/б ключом в словаре
print(l[1]) # OK
```

Последовательность:

```
for i in l:
        print(i) # OK
for in range(0,len(l)):
        print(l[i]) # OK
(a,b,c) = l # операция "привязки" (tie)
print (a); print(b); print(c)
(a,b) = (b,a)
```

Зачем? - очень удобно - объединять несколько возвращаемых ф-цией значений объединять в единое целое.

Одновременное присваивание - "бесплатно"

Структурные базисные типы - кортежи

Пример 2 - Go - тип времени компиляции- скобок нет! - кортеж существует только во время трансляции (операция tie() связана только статически)

```
dat, err := ioutils.ReadFile("myfile.txt")
if (err!=nil) {
        panic(e)
}
func foo () int, string { // кортеж - возвращаемое значение
    return (1, "string")
}
var i int
var s string
i, s =  foo()
i, _ = foo()
j,_:=foo()
```

Структурные базисные типы - кортежи

Пример 3 - C++ - тип времени компиляции (шаблон)

Появился еще в boost => C++11 с ограничениями

C++14 - вариадические шаблоны => полноценный кортеж

```
std::tuple<int, std::string> t (3, "string"); // std::pair
std:: tuple<int, double, std::string> d; // можно, т.к. для каждого типа есть КУ - int(), double()...
int j = std::get<0>(t);
std::string s = std::get<1>(t);
std::tuple<int, double> tt = std::make_tuple(1, 0.0);
// не read-only!!!
t = std::make_tuple(-1, "negative"); // если для типов есть operator =()
```

Можно сравнивать на ==, !=, если эти операции применимы к типам кортежа

Операция "привязки" - явная - std::tie()

```
tuple <int, int, int, int> barCode(8, 13457, 12345, 3);
int numberSystem, manufCode, prodCode, check;
std::tie(numberSystem, manufCode, prodCode, check) = barCode;
```

std::ignore - аналог _

```
std::tie(numberSystem, manufCode, prodCode, std::ignore) = barCode;
```

Структурные базисные типы - кортежи

Пример 4: Кортежи в Swift - полноправные типы-значения (переменные, параметры функций, возвр. значения функций).

```
let t = (1.0, "text")
var tt = t
print (t.0)
print(t.1)
```

Полный аналог записи:

```
var adr = (street: "Tallinskaya", house: 9, bldg: 2, ZIP: "123458");
adr.street ... adr.house ... adr.ZIP
```

Операция привязки (видели в перечислимых типах)

```
var (ave, h, _, zip) = adr;
```

Структурные базисные типы - кортежи

Пример 4: Кортежи в Rust - тоже полноправные типы-значения (переменные, параметры функций, возвр. значения функций).

```
let t = (1.0, "text");
var tt = t;
println !("Первое значение: {}", t.0);
println!("Второе значение: {}", t.1);
fn reverse(pair: (i32, bool)) -> (bool, i32) {
        let (i, b) = pair; // а вот и привязка
        (b,i)
}
```

Структурные базисные типы - кортежи

Общий вывод - некоторый аналог записи.

- Удобно
- "Сахар"
- Тип времени компиляции vs полноценный тип-значение ?

Структурные базисные типы - множества и таблицы

Идейно - близки (множество <=> таблица из одних ключей).

Основные операции над множествами:

S<T>: set of T; x : T

in: S,T -> Boolean (x in s)

add: S, T -> S (с новым элементом x) (INCL(s,x)) S := S + [x];

remove: S,T ->S (без элемента x) ((EXCL(s,x))) S := S - [x];

+: S, S -> S (объединение)

*: S, S -> S (пересечение)

- : S, S -> S (разность)

Все это было уже в Паскале. НО: ограничения на T (мощность и дискретность) => множество Паскаля - просто битовые шкалы ОЧЕНЬ небольшого размера.

Интересна эволюция понятия "множество" в ЯП от Н.Вирта

Модула-2 - как в Паскале, но добавился стандартный тип, зависящий от реализации:

BITSET = SET OF [0..WORD_BITS-1];

Что собой представляют операции выше над типом BITSET с точки зрения машинных операций?

Структурные базисные типы - множества и таблицы

Идейно - близки (множество <=> таблица из одних ключей).

Основные операции над множествами:

S: set of T; x : T

in: S,T -> Boolean (x in s)

add: S, T -> S (с новым элементом x) (INCL(s,x))

remove: S,T ->S (без элемента x) ((EXCL(s,x)))

+: S, S -> S (объединение)

*: S, S -> S (пересечение)

- : S, S -> S (разность)

Все это было уже в Паскале. НО: ограничения на T (мощность и дискретность) => множество Паскаля - просто битовые шкалы ОЧЕНЬ небольшого размера.

Интересна эволюция понятия "множество" в ЯП от Н.Вирта

Модула-2 - как в Паскале, но добавился стандартный тип, зависящий от реализации:

BITSET = SET OF [0..WORD_BITS-1];

Что собой представляют операции выше над типом BITSET с точки зрения машинных операций?

Побитовые операции!!!! - то есть BITSET - средство манипулирования битами внутри маш.слова.

Оберон - нет диапазонов => тип Модулы 2 SET OF T теряет свой смысл. Остается только BITSET, который переименовывается SET. Это единственный множественный стандартный тип. Других множеств нет.

Структурные базисные типы - множества и таблицы

То есть настоящих множеств в ЯП от Н.Вирта нет, равно как нет таблиц.

Таблицы - главная операция - поиск значения по ключу.

T<K,V>

lookup: K,V -> ref V

Паскаль - игровой ЯП. А в других языках?

Статические ЯП vs динамические ЯП.

Эффективность или универсальность?

В общем случае противоречат друг другу.

Множества vs таблицы

Древовидные таблицы (ordered maps) vs хэш-таблицы (unordered maps)

Универсального и одновременно максимально эффективного нет.

Структурные базисные типы - множества и таблицы

Статические ЯП - общий подход - не в базис, а в стандартную библиотеку с большим разнообразием алгоритмов (так же, как и с динамическим массивами). Как правило - обобщенные реализации (C++, C#, Java)

C++:

set<K>, multiset<K> (древовидные реализации)

unordered_set<K>, unordered_multiset<K> (хэши) (C++11)

map<K,V>, multimap<K,V> (древовидные реализации)

unordered_map<K,V>, unordered_multimap<K,V> (хэши) (C++11)

Особенность map<K,V>:

m[k] - всегда возвращает std::pair<const K&,V&>&

Если элемента с ключом k нет, то он вставляется со значением V().

Ну и есть, конечно, bitset<N> - реализация множеств целых чисел в виде битовой шкалы

Структурные базисные типы - множества и таблицы

Статические ЯП - общий подход - не в базис, а в стандартную библиотеку с большим разнообразием алгоритмов (так же, как и с динамическим массивами). Как правило - обобщенные реализации (C++, C#, Java)

C# (.Net)

Есть интерфейс - IDictionary<K,V> и обобщенные классы

- Dictionary<K,V>: IDictionary<K,V>
  с конструкторами:
  - public Dictionary()
  - public Dictionary(IDictionary d)
  - public Dictionary(int capacity)
- SortedDictionary<K,V>: IDictionary<K,V>
- Hashtable<K,V>: IDictionary<K,V>
  с конструкторами:
  - public Hashtable()
  - public Hashtable(IDictionary d)
  - public Hashtable(int capacity)
  - public Hashtable(int capacity, float loadFactor)

- SortedList<K,V>
- Set<T>
- SortedSet<T>

Структурные базисные типы - множества и таблицы

Статические ЯП - общий подход - не в базис, а в стандартную библиотеку с большим разнообразием алгоритмов (так же, как и с динамическим массивами). Как правило - обобщенные реализации (C++, C#, Java)

Java - множество классов-коллекций (плюс этого подхода) - стандартная библиотека (JDK) + guava, apache, trove, gs

Рассмотрим только стандартные классы и интерфейсы

Множества

Структурные базисные типы - множества и таблицы

Java JDK - Множества

Интерфейсы

Set<T> - общий интерфейс

SortedSet<T> implements Set<T> - элементы отсортированы (в каком-либо ! порядке)

NavigableSet<T> implements SortedSet <T> - двунаправленный обход

Классы

HashSet<T> implements Set<T>

LinkedHashSet<T> extends HashSet<T> implements Set<T> - хэш, прошитый однонаправленным списком

TreeSet<T> extends NavigableSet<T> - использует дв. деревья(RB trees)

EnumSet<T extends Enum<E>> - битовая шкала для перечислимых типов

А просто аналог bitset<N>?

Структурные базисные типы - множества и таблицы

Java JDK - Множества

Интерфейсы

Set<T> - общий интерфейс

SortedSet<T> implements Set<T> - элементы отсортированы (в каком-либо ! порядке)

NavigableSet<T> implements SortedSet <T> - двунаправленный обход

Классы

HashSet<T> implements Set<T>

LinkedHashSet<T> extends HashSet<T> implements Set<T> - хэш, прошитый однонаправленным списком

TreeSet<T> extends NavigableSet<T> - использует дв. деревья(RB trees)

EnumSet<T extends Enum<E>> - битовая шкала для перечислимых типов

А просто аналог bitset<N>?

Конечно есть - но не как обобщенный класс, а конкретный:

java.utils.BitSet s = new java.utils.BitSet(32);

Структурные базисные типы - множества и таблицы

Java JDK - таблицы

Интерфейсы

Map<K,V> - общий интерфейс

SortedMap<K,V> implements Map<K,V> - элементы отсортированы (в каком-либо ! порядке)

NavigableMap<K,V> implements SortedMap <K,V> - двунаправленный обход

Классы

HashMap<K,V> implements Map<K,V>

Linked HashMap<K,V> extends HashMap<K,V> implements Map<K,V> - хэш, прошитый однонаправленным списком

TreeMap<K,V> extends NavigableMap<K,V> - использует дв. деревья(RB trees)

EnumMap<K extends Enum<K>, V> - массив для перечислимых типов (использует raw-значения)

Частные реализации:

WeakHashMap<K,V> implements Map<K,V> - для слабых ссылок на ключ (но не значение!)

IdentityHashMap<K,V> implements Map<K,V> - идентичность вместо равенства - для эффективности

Структурные базисные типы - множества и таблицы

Динамические ЯП - "под рукой" (в базисе) всегда некоторый минимум. + библиотеки

JavaScript - объект - это хэш-таблица свойств (ключ - любая строка)

```
var o = {}
o.prop = 1
o.f = function(x) { return x+1 }
o.a = [1,2,3]
arr = []
for (p in o)
        arr.push(p)
```

```
var o = {"prop": 1, "f": function(x) { return x+1 }, "a": [1,2,3]}
```

JSON - "JavaScript Object Notation" ("открыт" Д.Крокфордом)

JSON.parse(str) <=>JSON.stringify(obj)

В ряде случаев вытеснила XML как формат обмена информацией между веб-клиентом и веб-приложением (и не только)

В настоящее время - часть библиотеки практически в любом языке

Структурные базисные типы - множества и таблицы

Динамические ЯП - "под рукой" (в базисе) всегда некоторый минимум. + библиотеки

JavaScript - ES-2015 - встроенные объекты Set, Map, WeakSet, WeakMap

Set и Map - любые ключи (с проверкой на равенство).

WeakSet, WeakMap - хранят слабые ссылки на ключи

Python - с самого начала содержал словари - dictionaries - хэш-таблицы с произвольными ключами и произвольными значениями (почему произвольными?)

Ключи - read-only объекты (типы-значения, кортежи, строки)

Структурные базисные типы - множества и таблицы

Динамические ЯП - "под рукой" (в базисе) всегда некоторый минимум. + библиотеки

Python - с самого начала содержал словари - dictionaries - хэш-таблицы с произвольными ключами и произвольными значениями (почему произвольными?)

Ключи - read-only объекты (типы-значения, кортежи, строки)

```
phones = {
        "GIG": "999-99-99",
        "Igor": "888-88-88"
}
phones["GIG"] = "123-45-67"
phones["Igor"] = [1, 2, -10] # а никто и не обещал про phones
```

Множества - как словари, но без значений - только ключи

```
set = {1, 2, 4, 10, "GIG"}
```

А что может быть ключом? Что-то неизменяемое, что имеет методы __hash()__ и __equ(obj)__ ("утиная типизация")

Структурные базисные типы - строки

Строка - структура для хранения (плоского) текста (то есть последовательность символов).

В чем отличия от массива символов?

Операции.

len(s) - длина (может отличаться от вместимости - capacity)

substr(s,sub) (charAt(s, ch)) - поиск подстроки (отдельного символа)

s1 + s2 - конкатенация

s1 <=> s2 - сравнение

s1:= s2 - копирование

Для строк - критичны (часто), для массивов - редко (часто не входят в базис).

Большое количество специфичных операций (не сравнить с массивами)...

В то же время ВАЖНЕЙШАЯ операция над массивами - индексирование

s[i] - индексирование (прямой доступ)

в массивах - всегда ref ValueType

В строках - много зависит от языка - далее обсудим

reverse() - для массивов практически везде, для строк - практически нигде...

Структурные базисные типы - строки

Строка - структура для хранения (плоского) текста (то есть последовательность символов).

В чем отличия от массива символов?

Операции.

.....

Динамичность (=> для текста - динамические строки)

Неизменяемость (для эффективности частого копирования и сравнения) (C#, Java, Python, JavaScript, Rust....) Swift - String и NSString: let s = "immutable"; var ss = "mutable"

Ну и наконец:

Специфика и важность текста.

Все современные ЯП за исключением C++ содержат string в базисе! ср. со словарями и множествами.

Почему не в стандартной библиотеке?

Не все операции можно реализовать средствами языка.

Аналогия с классами-обертками и некоторыми другими "встроенными" в язык.

Про эти классы транслятор "знает" что-то индивидуальное и неприменимое к другим классам из станд. библиотеки.

(C# - System, Java - java.lang)

Фактически C++ единственный достаточно мощный язык, чтобы использовать динамические строки из стандартной библиотеки

Представление текста в ЯП

Текст состоит из символов (литер). А что есть символ? Зависит от алфавита, точнее от используемого набора символов (charset), который основан, в первую очередь, на национальном алфавите и на международном алфавите.

Поэтому первая проблема при представлении текста - как представить отдельную литеру (тип CHARACTER), вторая проблема - как представить последовательность литер.

50-е гг. Вначале использовались кодировки BCD (Binary Coded Decimal) - 48 символов - мы BCD уже вспоминали - когда?

https://en.wikipedia.org/wiki/BCD_(character_encoding)

Потом эта кодировка расширялась (не заменялась) - появился BCDIC (Binary Coded Decimal Interchange Code) - каждый символ кодировался 6 битами.

Упаковка литер в маш.слово. FORTRAN - все идентификаторы <= 6 символов - это ограничение перешло на компоновщики. ОЧЕНЬ долго продержалось...

Почему 6? Внимание - вопрос - угадайте размер машинного слова в компьютере IBM-704/709/7090.

1961 - IBM/360 - адресуемая ячейка - байт, маш. слово - 4 байта.

Зачем байт - для хранения КОДА символа (от 0-255) в кодировке Extended BCDIC (EBCDIC). Туда "влезали" управляющие символы, ВЕСЬ английский алфавит (a-z, A-Z), цифры, знаки пунктуации - ну и немного места под что-то другое (например, нац. алфавиты типа кириллицы).

Сам код - ужасный - куча неплотных участков (даже для базового алфавита), и понятно почему...

Но - возникло понятие "кодовой страницы" и семейства "кодовых страниц". EBCDIC - семейство однобайтовых кодовых страниц. В частности советский стандарт - ДКОИ-8. Позже появились двухбайтовые КС и т.д.

https://en.wikipedia.org/wiki/Code_page#EBCDIC-based_code_pages

Вместо EBCDIC-кодов в США ввели более удачный код - ASCII-7 (7 битов на символ).

Дальнейшие распространенные кодировки используют ASCII-7 как основу (первые 128 кодов). DEC PDP/11, IBM PC (MS DOS, IBM OS/2), MS Windows, UNIX...

Проблема - что делать со старшей кодовой страницы. 128 кодов для одного алфавита - нормально (так????), а для нескольких уже тесно.

Только для русского языка - ДКОИ-8, КОИ-8Р, СР-866, СР-1251 (последние 3 - на базе ASCII-7), для кириллицы и основанных на кириллице алфавитах есть и еще...

Представление текста в ЯП

Для западно-европейских алфавитов - ISO-Latin1 (еще Latin2-....Latin5), для остальных - свои однобайтовые кодовые страницы (SBCS- Single-Byte Character Set).

Как представить текст:

Французское слово café переводится на русский как кафе

Никак в SBCS (просто потому, что смешиваются ISO-Latin1 и кириллица - нет таких кодовых страниц). Другой пример - Turkish

Для алфавитов,превышающих по числу 255, были разработаны системы кодировки DBCS (Double-Byte Character Set) - например для китайского, и MBCS (MultiByte Character Set) - например, ShiftedJIS для японского.

Путаница, естественно, только усугубилась. Но это не было проблемой до 80-х гг 20 века для основных разработчиков ПО.

Проблемы появились из-за развития использования ВТ в странах, не покрываемых ISO-Latin1.

Выход - стандартизация.

Международный консорциум UNICODE (1991) разработал стандарт, определяющий в настоящее время базовые правила представления произвольного текста в компьютерах. Значительная часть стандарта Unicode отражена в стандарте ISO-10646 (немного разная терминология)

Представление текста в ЯП. Юникод.

Юникод состоит из двух основных частей:

- универсальный набор символов (universal character set-UCS)
- набор кодировок (UCS Transformation Format - UTF), которые определяют способ представления символов на компьютере

Также определен ряд правил нормализации текста (нужны для совместимости с установившимися способами представления текста).

Примеры - каноническое расширение и каноническое сужение.

Пример проблемы нормализации - некоторые символы можно представлять двояко - как один символ (é - U+E9), и как последовательность символов, первый из которых основной (e - U+65), второй - комбинированный модифицирующий диакритический (комбинированный акут ´ - легкое ударение- сам по себе не является отдельным символом - U+301). Для русского - тоже иногда актуально (ударения, й, ё).

Отступление о реверсе строк

Представление текста в ЯП. Юникод.

Юникод состоит из двух основных частей:

- универсальный набор символов (universal character set-UCS)
- набор кодировок (UCS Transformation Format - UTF), которые определяют способ представления символов на компьютере

Универсальный набор символов:

- стандартное имя символа
- уникальный номер этого символа - кодовая точка (code point)

Пример:

CYRILLIC CAPITAL LETTER A – заглавная буква А кириллического алфавита CYRILLIC SMALL LETTER ZHE – строчная буква Ж кириллического алфавита

Запись кодовой точки: U+*hex_code*

CYRILLIC CAPITAL LETTER A - U+0410

CYRILLIC SMALL LETTER ZHE - U+436 (нули можно и опускать)

В ряде ЯП с представлением строк в Юникоде внутри строк допускаются экранированные символы в стиле C, но с Юникодом:

"This is CYRILLIC CAPITAL LETTER A - \u410!"

Представление текста в ЯП. Юникод.

Юникод состоит из двух основных частей:

- универсальный набор символов (universal character set-UCS)
- набор кодировок (UCS Transformation Format - UTF), которые определяют способ представления символов на компьютере

Некоторые особенности UCS:

для национальных алфавитов и других исторически выделившихся последовательностей символов предусмотрены отдельные диапазоны:

первые 128 кодовых точек (U+0 ....U+7F) - ASCII-7

первые 255 кодовых точек (U+0 ....U+FF) - ISO-Latin1 (предыдущая строка избыточна)

....

Кириллица - аж несколько диапазонов:

Базовый - U+0400 - U+04FF

Дополнение - U+0500 - U+052F

Расширенная кириллица - C - U+1C80 - U+1C8F

Расширенная кириллица - A- U+2DE0 - U+2DFF

Расширенная кириллица - B - U+A640 - U+A69F

Ну и глаголица "до кучи" - U+2C00 - U+2C5F

Надо смотреть на это богатство (без кавычек), например:

https://unicode-table.com/ru/blocks/

Представление текста в ЯП. Юникод.

Юникод состоит из двух основных частей:

- универсальный набор символов (universal character set-UCS)
- набор кодировок (UCS Transformation Format - UTF), которые определяют способ представления символов на компьютере

Первый вариант Юникода - 65535 кодов (помещаются в 2 байта) - иногда (не совсем точно) называют UCS-2. Основные алфавиты мировых языков и ряд важных диапазонов.

Дальнейшие версии - расширяют UCS.

Зачем?

Искусственные языки и алфавиты

Китай

Новые символы (пример - смайлики, эмоджи, гос. флаги...)

....

Все кодовые точки из Юникода разбиты на непрерывные сегменты длиной 65536 - проекции кодовых точек.

Плоскость 0 - U+0000 - U+FFFF - основная многоязычная плоскость - Basic Multilingual Plane (BMP)

Плоскость 1 - U+10000 - U+1FFFF - дополнительная многоязычная плоскость

Плоскость 2 - U+20000 - U+2FFFF - дополнительная идеографическая плоскость

Плоскость 3 - U+20000 - U+2FFFF - еще одна (третичная) идеографическая плоскость

Плоскость 3 - Плоскость 14 - пока не используются

Плоскость 15 - U+E0000 - U+EFFFF - специализированная плоскость

Плоскость 16 - U+F0000 - U+FFFFF - частная (private) плоскость A

Плоскость 17 - U+100000 - U+10FFFF - частная (private) плоскость A

Представление текста в ЯП. Юникод.

Юникод состоит из двух основных частей:

- универсальный набор символов (universal character set-UCS)
- набор кодировок (UCS Transformation Format - UTF), которые определяют способ представления символов на компьютере

17 плоскостей (почему не 32 или 33?)

Главная - BMP. Определена в 1991 году. Ассоциировалась (неправильно!) с Юникодом.

Но кодовая точка - это только число. Как это число представляется в памяти компьютера? - это вопрос ***кодировки***