Представление текста в ЯП. Строки

Swift - самое радикальное решение.

char - это что? То, что отображается на экране (бумаге и т.п.) - extended grapheme cluster

string - последовательность из char

Нет прямого доступа! Нет вообще индексации по целым!

Есть специальный тип - индекс символа в строке (с итерацией)

Кодировки (8,16,32)- доступны через View - свойства строк.

Вывод - очень непривычно, но самый "честный" с точки зрения отображения путь

Представление текста в ЯП. Строки

```
let eAcuteQuestion = "Voulez-vous un caf\u{E9}?"
// "Voulez-vous un café?" используем LATIN SMALL LETTER E и COMBINING ACUTE ACCENT
let combinedEAcuteQuestion = "Voulez-vous un caf\u{65}\u{301}?"
if eAcuteQuestion == combinedEAcuteQuestion {
  print("Эти строки считаются равными")
}
for codeUnit in eAcuteQuestion.utf16 {
    print("\(codeUnit) ", terminator: " ")
}
print("")
for codeUnit in combinedEAcuteQuestion .utf16 {
    print("\(codeUnit) ", terminator: " ")
}
print("")
for scalar in combinedEAcuteQuestion .unicodeScalars {
   print("\(scalar.value) ", terminator: " ")
}
print("")
let leaf = "\u{1f342}";
print( leaf)
for scalar in leaf.unicodeScalars {
   print("\(scalar.value) ", terminator: " ")
}
```

Представление текста в ЯП. Строки

```
let greeting = "Guten Tag!"
 greeting[greeting.startIndex]
 // G
 greeting[greeting.index(before: greeting.endIndex)]
 // !
 greeting[greeting.index(after: greeting.startIndex)]
 // u
 let index = greeting.index(greeting.startIndex, offsetBy: 7)
 greeting[index]
 // a
for index in greeting.indices {
 print("\(greeting[index]) ", terminator: " ")
 }
// Выведет "G u t e n T a g !"
```

Представление текста в ЯП. Строки

JavaScript - типа char нет.

Есть String (неизменяемая) - в настоящее время - UTF32

(сравним с Python)

Задачи на строки из вариантов прошлых лет

2017 - задача 1

2018 - задача 2

Операторный базис в процедурных ЯП

Отступление о терминологии (expression, operator, statement - выражение, операция, оператор)
Почему? Операторные схемы Ляпунова - 1956, Алгол - 1960
V := Expression
I/O - в станд. библиотеку
Что осталось? Передача управления (control flow)
Ранние ЯП - блок-схемы (flow-charts)
см. картинки
$\Rightarrow$ главная управляющая конструкция - оператор перехода
Fortran:

```
GOTO 5
IF (N) 5,10,15
ASSIGN 555 TO M
GOTO M
GOTO (10,20,30,40, 50) i
```

Ну и наконец...

```
CALL proc (20, 30, 40)
......
SUBROUTINE proc (*,*,*)
......
RETURN k
END
```

Операторный базис в процедурных ЯП

DS-programs (Dish-of-Spaghetti)

или "спагетти-код" (взять листинг и линиями отрисовать все переходы - "откуда-куда") - неоправданные и запутывающие

алгоритм передачи управления

Примеры "спагетти-кода"

Операторный базис в процедурных ЯП

1967 - Э.Дейкстра - "Goto Statements Considered Harmful"

Далее - Дал, Дейкстра, Хоар - "Structured Programming"

Структурное программирование

Практически немедленное принятие всем сообществом

$\Rightarrow$практически полное единообразие управляющих структур в ЯП с конца 60-х гг 20 века

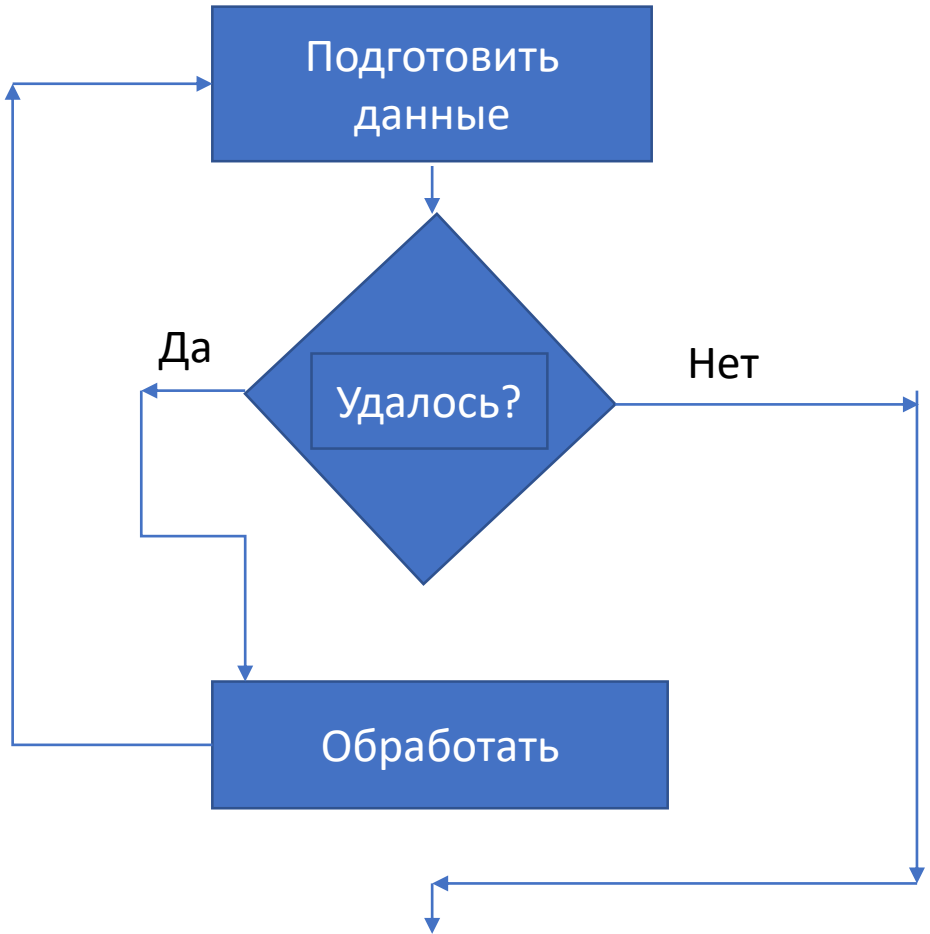
Управляющие структуры в процедурных ЯП

Паскаль:
- последовательность : S1; S2
- развилки: if D then S; if B then S1 else S2; case E of .... end;
- циклы: while B do S; repeat S1; ... until B; for v:=e1 to e2 do S
- goto label; (только локальные и ограниченные)

C:
практически аналогичные (слегка "выбивается" switch)
+ добавлены "заменители" goto:
return; break; continue; - они важны!
Д.Кнут - 1974 - "Структурное программирование с операторами перехода" ("Structured programming with goto statements")

Управляющие структуры в процедурных ЯП

Типичная схема обработки данных:



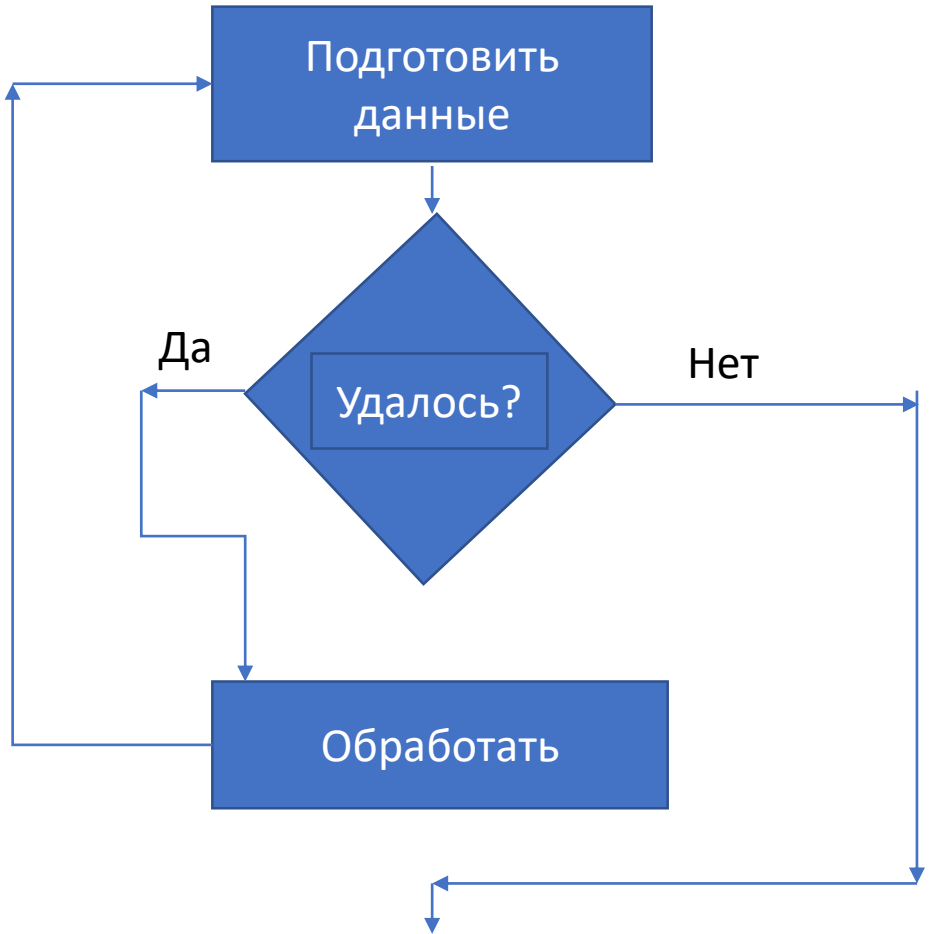
Как это записать в чисто структурных операторах?

```
Prepare(data);
while (DataReady(data)) {
  Process(data);
  Prepare(data);
}
```

Очень плохо!!!

Управляющие структуры в процедурных ЯП

Типичная схема обработки данных:



Как это записать в чисто структурных операторах?

```
for (;;) {
   Prepare(data);
   if  (!DataReady(data))
        break;
   Process(data);
}
```

Выход из середины цикла - вполне отвечает СТРУКТУРЕ алгоритма

Управляющие структуры в процедурных ЯП

Паскаль:
- последовательность : S1; S2
- развилки: if D then S; if B then S1 else S2; case E of .... end;
- циклы: while B do S; repeat S1; ... until B; for v:=e1 to e2 do S
- goto label; (только локальные и ограниченные)

Модула 2, Оберон:
- последовательность : S1; S2
- развилки
- циклы while, repeat, for (нелегкая судьба) +

LOOP S_1;..... S_N END
- вместо goto - RETURN (в подпрограммах) и EXIT (только в LOOP циклах)
- goto нет совсем....

Управляющие структуры в процедурных ЯП

Итак:

последовательность, развилки, циклы while-repeat-for + заменители goto

Вариации на основную тему

Зачем оператор выбора (case - switch)?

- интегрированность с перечислениями и размеченными объединениями
- ОЧЕНЬ эффективно компилируется (Фортран - вычисляемый GOTO, язык ассемблера - JMP DWORD_PTR jmpTable[EDI] -
- см. файл Switch_translation_scheme.cpp)

Где это не важно?

Управляющие структуры в процедурных ЯП

Питон - нет его совсем - а как же быть?
Одна из альтернатив - замена на последовательный выбор ("подарок" - трехсторонне сравнение - тоже синтаксический сахар)

```
if 0 <= i <=3 or 5<=i <= 7:
        S1
elif i==8:
        S2
else:
        S3
```

(какой switch или case? Какой вариант эффективнее?)
Более общая замена - словарь с (анонимными) функциями
А почему бы и не добавить переключатель или выбор (как в JavaScript?) - чтобы не обманывать программистов.

Управляющие структуры в процедурных ЯП

Переключатель:
C# (расширен на string):

```
switch (country) {
case "UK": capital = "London"; break; // опустить break нельзя...
case "RF": capital = "Moscow"; break;
case "US": capital = "Washington"; break;
default: capital = "Some other city"; break;
}
```

Как реализовать? (goto по таблице - не выйдет) - через последовательный выбор - if-then-else if - then - else if..... - плохо (почему?)
При количестве альтернатив >2 компилируется в словарь, ключами служат строки, значениями - метки перехода.

Управляющие структуры в процедурных ЯП

Итак:
последовательность, развилки, циклы while-repeat-for + заменители goto
Вариации на основную тему
Цикл for - зачем он?
A-60 - Паскаль.....- перебор элементов массива
C: самая общая форма цикла (for (e1;e2;e3) S)
Сейчас - циклы-итераторы (в Питоне - только такие циклы + while, в Go - только две формы for - как в C + итераторы)
Почему? Интеграция с любыми последовательностями (массив - частный случай)

Управляющие структуры в процедурных ЯП

Циклы-итераторы
Питон
Go
C#
Ну и даже в Java и C++
Современная тенденция - интеграция базисных конструкций ЯП с пользовательскими типами данных
Что нужно? Три операции

- Инициализировать итератор
- Продвинуть итератор на след. элемент с проверкой
- Получить текущий элемент

Управляющие структуры в процедурных ЯП

Циклы-итераторы в языке C#

```
foreach (var x in Container) {
   // x - поочередно принимает значения последовательных элементов в контейнере
}
```

Пример:

```
int [] arr = new int[]{1,2,3,4,5,6,7,8,9};
foreach (int x in arr)
   sum += x;
Console.WriteLine("Sum=" + sum.ToString());
```

Что требуется от Container?

Три операции:

- Инициализировать итератор
- Продвинуть итератор на след. элемент с проверкой
- Получить текущий элемент

То есть контейнер (последовательность) должен удовлетворять некоторому КОНТРАКТУ

Управляющие структуры в процедурных ЯП

То есть контейнер (последовательность) должен удовлетворять некоторому КОНТРАКТУ - есть специальная конструкция - interface
Точнее - контрактов (интерфейсов) - два - первый (IEnumerable) возвращает итератор:

```
interface IEnumerable {
        IEnumerator GetEnumerator();
}
```

второй (IEnumerator) - постулирует набор операций, которые должен реализовывать каждый итератор:

```
interface IEnumerator {
  void Reset(); // инициализировать итератор для новой серии итераций
  bool MoveNext(); // продвинуть итератор на след. элемент с проверкой
  Object Current; get; // получить текущий элемент (без изменения самой коллекции)
}
```

О реализации - позже, когда будем говорить о сопрограммах.

Управляющие структуры в процедурных ЯП

```
interface IEnumerable {
            IEnumerator GetEnumerator();
}
interface IEnumerator {
  void Reset(); // инициализировать итератор для новой серии итераций
  bool MoveNext(); // продвинуть итератор на след. элемент с проверкой
  Object Current; get; // получить текущий элемент (без изменения самой коллекции)
}
foreach (Object x in Container) {
  .....
} //это "синтаксический сахар" для следующей конструкции:
IEnumerable it = Container.GetEnumerator();
it.Reset();
while (it.MoveNext()) {
    Object x = it.Current; ....
}
```

Управляющие структуры в процедурных ЯП

В языке Java тоже есть похожие интерфейсы (Iterable и Iterator)

```
interface Iterable {
        Iterator iterator();
}
interface Iterator {
        bool hasNext(); // есть ли еще элементы?
        Object next();   //  возвращает очередной элемент и переходит на след. элемент
        void remove(); // расширяет семантику по сравнению с C#
} // В C# более логичные, на мой взгляд, названия.....
```

next() и hasNext() совместно реализуют семантику MoveNext() и Current

А в 2005 появился и "синтаксический сахар" (цикл-итератор):

```
for (Object x: Container) { x.....}
Iterator it = Container.iterator();
while (it.hasNext()) {
    Object x = it.next(); ....
}
```

Управляющие структуры в процедурных ЯП

Обобщенные интерфейсы (параметризованные типом элемента):

```
interface IEnumerable <T> {
        IEnumerator<T> GetEnumerator<T>();
}
interface IEnumerator <T> {
  void Reset(); // инициализировать итератор для новой серии итераций
  bool MoveNext(); // продвинуть итератор на след. элемент с проверкой
  T Current; get; // получить текущий элемент (без изменения самой коллекции)
}
List<string> txt;
foreach (int x in txt) { ..... }// ошибка трансляции
```

Раньше (до обобщений):

```
List txt;   txt.Add("line1"); txt.Add("line1");
foreach (Object o in txt) {
  int  x = (int)o; // ошибка во временя выполнения
}
```

В Java 2005 также появились обобщения и обобщенные (параметризованные) интерфейсы

Управляющие структуры в процедурных ЯП

Язык Go (уже видели циклы-итераторы для строк):

```
// если это коллекция, то к ней применимо ключевое слово range
for k, v := range C { // итератор по коллекции
                      // k - ключ, v - значение
                      // если массив, то k - индекс, v - значение
                      // если строка, то k - позиция от начального байта, v - rune
                      (значение в unicode)
// частный случай коллекции - map (входит в базис), k - ключ, v - значение}
// map[тип ключа] тип значения
d map[string] string
for _, v := range C { // фактически, это цикл foreach
}
```

Управляющие структуры в процедурных ЯП

Язык Питон: цикл for - только итератор
for x in coll:
    ...x...
coll - это "нечто", что имеет итератор, либо coll - генератор
Отличия итератора и генератора - итератор "проходит" по существующей структуре и выдает следующий элемент, генератор - "генерирует" следующее значение (и насколько глубока эта разница?)
Пример: как же просто "пройти" от i до j? (for (index = i; index < j; index++) ...)
range(i,j) - в Питоне < 3.xx - это список (массив) [i, i+1,.... j-1]
xrange(i,j) - в Питоне < 3.xx - это генератор последовательности i, i+1,.... j-1 (без хранения всей структуры в памяти)
Питон 3.xx - range(i,j) - генератор, xrange(i,j) - отсутствует
for index in range(i,j): .....

Управляющие структуры в процедурных ЯП

Язык Питон: цикл for - только итератор

```
for x in coll:
        ...x...
for index in range(i,j):
        ...index....
```

Использование цикла-итератора для генерации списков (list comprehension):

```
my_xrange = [x for x in range(i,j)]
```

Шаблон математического определения некоторого множества S1 через предикат P над другим множеством S

$S1 \equiv \{x \in S \mid P(x)\}$

Но: S1 - это список, а вот S - произвольная "последовательность"

```
S1 = [x for x in S if P(x)]
```

Управляющие структуры в процедурных ЯП

Язык Питон: использование цикла-итератора для генерации списков (list comprehension):

Почему не стоит "упростить": вместо S1 = [x for x in S if P(x)]

использовать:

S1 = [for x in S: if P(x)]

Потеря и общности, и мощности:

L = [x*x for x in range(0,10)]

S1 = [(x,x // 2) for x in range(0,100) if x%2 == 0] # // - целочисл. деление

S2 = [(y,x) for (x,y) in S1]

Управляющие структуры в процедурных ЯП

Вывод: набор управляющих структур для (объектно-)императивных ЯП стабилен практически полвека - циклы while и repeat, циклы-итераторы, развилки общего вида (if-then; if-then-else), для компилируемых ЯП - оператор дискретного выбора (переключатель), заменители goto (return, break, continue).
Есть специализированные расширения, как правило связанные с добавлением новых парадигм в императивную, например, параллельные процессы (задачи) в языке Ада => операторы select и accept, квазипараллельные потоки и неименованные каналы в языке Go=> операторы select, defer и др., сопрограммы и потоки (C#) => оператор await и другие.

Средства развития в объектно-императивных ЯП

Развитие <=> создание новых абстракций

- Абстракция операции - функция
- Абстракция оператора - процедура

подпрограммы

- Абстракция потока управления - сопрограмма, поток, процесс
- Абстракция данных - новые типы данных (например, классы)

+ средства защиты (поддержки) создаваемых абстракций

инкапсуляция

модули

обработка ошибок

обобщения

и многое другое....

Средства развития в ОИ языках - подпрограммы

Подпрограмма - параметризованная абстракция алгоритма

Два аспекта - передача (поток) данных и передача (поток) управления.

Вначале сосредоточимся на первом аспекте - передаче данных.

Аргументы (формальные параметры) подпрограммы - при определении п/п

Фактические параметры - ДИНАМИЧЕСКИ связываются с формальными параметрами при вызове п/п (вспомним понятие времени связывания)

Период связывания - время работы подпрограммы.

Функция - подпрограмма, вычисляющая значение по значениям своих аргументов

Процедура - подпрограмма, меняющая значение своих аргументов (если меняются еще и глобальные переменные, то имеет место побочный эффект)

Средства развития в ОИ языках - подпрограммы

Передача данных в подпрограммах

Как отличаются функции от процедур?

Главный аспект - изменение (фактических) параметров.

Если в функции - нельзя, то (забывая про побочный эффект) использовать функцию осмысленно ТОЛЬКО в выражениях.

Если можно (изменять параметры и побочный эффект), то допустимо понятие "оператор-выражение":

funct(1,2,x) - как отдельный оператор.

Статические языки - дело "вкуса" (C vs Паскаль, Green Ada), динамические ЯП - второй случай (почему?)

Средства развития в ОИ языках - подпрограммы

Передача данных в подпрограммах

Связывание фактических и формальных параметров.

Inout - семантика - абстрактная модель - формальные параметры разделяются на три вида:

- in - нельзя менять их значение (=>нельзя менять и значение соответствующих фактических), но можно использовать => фактические параметры должны быть определены в момент вызова
- out - можно менять их значение, при этом значение фактических тоже изменится (когда именно??? - не уточнено). Использовать это значение можно только после первого присваивания (поэтому фактические параметры могут иметь неопределенное значение в момент вызова)
- inout - комбинация первых двух - можно использовать значения формальных параметров => фактические параметры должны быть определены в момент вызова, и можно менять их значение, при этом значение фактических тоже изменится (когда именно??? - не уточнено).

Средства развития в ОИ языках - подпрограммы

Передача данных в подпрограммах

Связывание фактических и формальных параметров.

Inout - семантика - абстрактная модель - формальные параметры разделяются на три вида: in, out, inout

Ада 83 - программист использует только эту модель, а компилятор должен выбрать какой-то конкретный способ связывания формальных и фактических параметров (исходя прежде всего из эффективности)

Способ связывания <=> способ "передачи параметров".

Спойлер: подход в Ада 83 провалился, все (объектно-)императивные современные ЯП используют ЯВНУЮ спецификацию способа передачи параметров (из которого вытекает inout-семантика)

Средства развития в ОИ языках - подпрограммы

Передача данных в подпрограммах

Связывание фактических и формальных параметров.

Способы передачи параметров:

1. по значению
2. по результату
3. по значению/результату
4. по адресу (ссылке)
5. по имени