Функциональная парадигма (язык Лисп)

Чтобы больше почувствовать язык, напишем собственную реализацию reverse

Используем функцию append:

(append S1 S2) => список - конкатенация S1 и S2

(append '(a b c) '(d e)) => (a b c d e)

Как определять новые функции?

Спец. функция defun

(defun f arglist S) - ничего не вычисляется!

f - символ, получающий функциональное значение

arglist - список символов - аргументов (нельзя подсписки и атомы)

S - список - тело функции.

(defun plus13(x) (+ x 13))

(plus13 10) => 23

Как происходит вызов?

Вычисляется аргумент, и вычисленное значение отождествляется с формальным параметром-символом из определения plus13

Функциональная парадигма (язык Лисп)

Свой вариант reverse ("наивный"):

```
(defun reverse1(s)
  (if (null s)
     s
     (append (reverse1 (cdr s)) (cons (car s) ()))
  )
)
```

Что будет, если убрать cons?

```
(append (reverse1 (cdr s)) (car s))
```

Функциональная парадигма (язык Лисп)

В чем "наивность"?

В квадратичности - так как append "пробегает" до конца первого аргумента на каждом шаге рекурсии (а их - N)

Решение - ввести вспом. функцию "сдвига" и накапливать на каждом шаге результат. "Сдвиг" головы левого аргумента в голову правого

(**1** 2 3 4) () => (2 3 4) (**1**)

(**2** 3 4) (1) => (3 4) (**2** 1)

(**3** 4) (2 1) => (4) (**3** 2 1)

(**4**) (3 2 1) => () (**4** 3 2 1) - в правом списке накоплен реверс исходного списка

Функциональная парадигма (язык Лисп)

Этот прием называют введением накопительного параметра

(l - обрабатываемый список, r - накопительный параметр)

```
(defun shift (l r)
   ( if (null l)
        r
        (shift (cdr l) (cons (car l) r))
   )
)
(defun reverse1 (s) (shift s nil))
```

Функциональная парадигма (язык Лисп)

```
(defun shift (l r)
   ( if (null l)
        r
        (shift (cdr l) (cons (car l) r))
   )
)
(defun reverse1 (s) (shift s nil))
```

Попробуйте написать свой вариант append

Аппликативная парадигма (язык Рефал)

REFAL - **Re**cursive **F**unctions **A**lgorithmic **L**anguage

В.Ф.Турчин (СССР - США) - 1966-1968

Фактически - особая парадигма, ближе к ФП, но есть также поиск с возвратами, сопоставление с образцом, как в ЛП.

Основные операции - сопоставление с образцом и вызов функции

Функция имеет один (и только один) аргумент (поле ввода)

Поле ввода - последовательность символов(литер)

Тело функции - последовательность предложений вида:

$\alpha=\beta$

α - описывает образец

β - описывает то, во что преобразуется образец, если поле ввода удалось сопоставить образцу.

При успешном сопоставлении функция завершается, возвращая то, что получилось в результате применения правой части успешного образца.

При неудаче сопоставления переходим к следующему образцу

Что-то напоминает?

Аппликативная парадигма (язык Рефал)

В левой и правой частях могут кроме конкретных символов появляться S.i - сопоставитель с символом (литерой), E.j - сопоставитель с выражением (последовательностью литер с некоторой структурой). Номера вхождений i,j нужны для отличения различных вхождений.

S.1 везде в правиле сопоставления означает один и тот же символ

E.1 - одно и то же выражение

Пример функции, определяющей, является ли строка палиндромом (/; - экранированный символ конца правила)

```
Palindrome {
                = 'Yes' /;
  s.1           = 'Yes' /;
  s.1 e.1 s.1 = <Palindrome e.1 > /;
  e.1           = 'No' /;
}
```

Аппликативная парадигма (язык Рефал)

Пример с реверсом (Prout - функция вывода, Card - функция ввода строки)

```
$ENTRY Go
{
    = <Prout <Reverse1 <Card>>>/;
}
Reverse1
{
    s.1 e.1 = <Reverse e.1> s.1/;
     = ;
}
```

Попробуйте написать "Hello, world!" на Рефале

Логическая парадигма (язык Пролог)

Prolog - **Pro**gramming in **Log**ic - 1971 - А.Колмерое (A.Colmerauer) - Марсель

Идея - описать семантику задачи в терминах формул исчисления предикатов (в некоторой нормализованной форме)

Спецификация задачи = программа, решающая задачу

Логическая парадигма (язык Пролог)

Prolog - **Pro**gramming in **Log**ic - 1971 - А.Колмерое (A.Colmerauer) - Марсель

Идея - описать семантику задачи в терминах формул исчисления предикатов (в некоторой нормализованной форме)

Спецификация задачи = программа, решающая задачу

Слишком хорошо, чтобы быть на самом деле...

Но в ряде задач показал себя хорошо

Логическая парадигма (язык Пролог)

Скалярный базис - простые данные

- константы
  - атомы (=символы, но со строчной буквой вначале, sokrat...., а также литералы 'Yes, it is')
  - числа (123)
- переменные (=идентификаторы, но с прописной буквы, X,Y,Z...., либо _x)

Структурный базис - списки

[1,2,3], [], [X|Y]

Программа на Прологе = описание предикатов

Предикаты (в ФП -логические функции). Как произвольную функцию описать в виде предиката?

Добавить еще один аргумент - результат.

append(L1,L2,L3) - истина, если L3 - конкатенация списков L1 и L2

Логическая парадигма (язык Пролог)

Предикаты описываются двояко - факты и правила

Факт:

ИмяПредиката(список_константных_аргументов).

Пример:

man(sokrat).

reverse([],[]).

Правила

P(X1,X2,X3,....) :- P1(X1,....), P2(X1,...,Y,....), ...

:- означает импликацию, запятая в правой части - логическое "и"

Правила соединяются в единую формулу путем логической операции "или". Фактически получаем единую формулу - дизъюнкцию импликаций (каждое правило - хорновсая клауза, или хорновский дизъюнкт)

Логическая парадигма (язык Пролог)

Пример - силлогизм Сократа ("каждый человек смертен, Сократ - человек, значит он - смертен")

man(sokrat).

mortal(X):-man(X).

Нужен еще запрос - а что нас интересует (смертен ли Сократ?)

? - mortal(sokrat).

true.

Логическая парадигма (язык Пролог)

man(sokrat).

mortal(X):-man(X).

Нужен еще запрос - а что нас интересует (смертен ли Сократ?)

? - mortal(sokrat).

true.

А какие могут быть еще запросы?

А кто вообще смертен?

Логическая парадигма (язык Пролог)

man(sokrat).

mortal(X):-man(X).

Нужен еще запрос - а что нас интересует (смертен ли Сократ?)

? - mortal(sokrat).

true.

А какие могут быть еще запросы?

А кто вообще смертен?

? - mortal(X).

X = sokrat.

Логическая парадигма (язык Пролог)

Добавим еще людей:

man('Donald Trump').

man(sokrat).

mortal(x):-man(x).

? - mortal(X).

X = sokrat ; здесь интерпретатор ждет подсказки (. или ;)

X = 'Donald Trump'. а здесь он сам сообщает, что .... точка!

Логическая парадигма (язык Пролог)

Реверс:

```
reverse1([], []).
reverse1([X|Q], Z) :- reverse1(Q, Y), append(Y, [X], Z).
?-reverse1([1,2,3,4],_)
true.
```

Логическая парадигма (язык Пролог)

Реверс:

```
reverse1([], []).
reverse1([X|Q], Z) :- reverse1(Q, Y), append(Y, [X], Z).
?-reverse1([1,2,3,4],_)
true.
?-reverse1([1,2,3,4],_x)
_x = [4,3,2,1].
```

Логическая парадигма (язык Пролог)

Реверс:

```
reverse1([], []).
reverse1([X|Q], Z) :- reverse1(Q, Y), append(Y, [X], Z).
?-reverse1([1,2,3,4],_)
true.
?-reverse1([1,2,3,4],X).
X = [4,3,2,1].
?-reverse1(X, [1,2,3,4]).
X = [4,3,2,1].
```

Логическая парадигма (язык Пролог)

Реверс:

```
reverse1([], []).
reverse1([X|Q], Z) :- reverse1(Q, Y), append(Y, [X], Z).
?-reverse1([1,2,3,4],_)
true.
?-reverse1([1,2,3,4],X).
X = [4,3,2,1].
?-reverse1(X, [1,2,3,4]).
X = [4,3,2,1].
?-reverse1([1,2,3,4],[X|Y]).
```

Логическая парадигма (язык Пролог)

Реверс:

reverse1([], []).

reverse1([X|Q], Z) :- reverse1(Q, Y), append(Y, [X], Z).

Тоже наивная реализация

(сравните с лисповской...)

Как можно переписать в стиле лисповской реализации с функцией сдвига?

Предикаты - тоже функции

Логическая парадигма (язык Пролог)

F(X,Y,Z)=R <=> P_F(X,Y,Z,R)

То есть, первое правило:

reverse2(X,Y):-shft(X,[],Y).

shift - стандартный предикат Пролога, поэтому используем символ shft

Логическая парадигма (язык Пролог)

F(X,Y,Z)=R <=> P_F(X,Y,Z,R)

shft(L,R,Z) <=> "Z есть результат переноса L в R"

То есть, первое правило:

reverse2(X,Y):-shft(X,[],Y). // reverse2(X) = shift(X, ())

Базис вывода (базисный факт):

shft([],X,X).

Логическая парадигма (язык Пролог)

F(X,Y,Z)=R <=> P_F(X,Y,Z,R)

shft(L,R,Z) <=> "Z есть результат переноса L в R"

То есть, первое правило:

reverse2(X,Y):-shft(X,[],Y). // reverse2(X) = shift(X, ())

Базис вывода (базисный факт):

shft([],X,X). // shift((),X) = X для любых X

Общее правило вывода:

shft([H|T],R,Z):-shft(T,[H|R],Z).

Логическая парадигма (язык Пролог)

Главная проблема Пролога в том, что он не является "истинно" логическим языком. Процедурная семантика превалирует над денотационной.

Но в ряде проблемных областей хорошо себя показал.

Формализм DCG (Definitive Clause Grammars) - Ф.Перейра, Д.Уоррен - основан на Прологе, позволяет описывать реальные грамматики ЕЯ, реализующие сложные синтаксические и семантические связи.

Основные понятия ЯП

На этом закончим введение в базисные парадигмы программирования и поговорим об основных понятиях ЛЮБОГО ЯП

Являются ли понятия "оператор", "константа", "адрес" универсальными понятиями любого ЯП?

Основные понятия ЯП

На этом закончим введение в базисные парадигмы программирования и поговорим об основных понятиях ЛЮБОГО ЯП

Являются ли понятия "оператор", "константа", "адрес" универсальными понятиями любого ЯП?

Конечно, нет!

А вот понятие "переменная"?

Формально - нет. Но фактически присутствует в большинстве ЯП независимо от парадигмы, но смысл - разный!

Имп.ПП - абстракция места в памяти (оператор := в алголоподобных ЯП)

ФПП - ближе к математическому понятию

Основные понятия ЯП

Основные понятия ЛЮБОГО ЯП. Их всего три:

- Данные
- Операции
- Связывание (менее очевидное понятие)

Программа <=> Данные. Операции - и есть <=>

Для каждого ЯП свой (уникальный?) набор операций и данных

Дуализм данных и операций

Примеры:

- операция (вычисления или считывания?), возвращающая длину строки, в ряде ЯП
- мемоизация значений вместо вычисления
- понятие "свойства" в ЯП

Свойство как понятие ЯП

Свойство (property) - C#, VB, Delphi,... Пример из C#:

```
class ClsWithProperty
{
        int _x = 0;
        public int X
        {
           get { return _x; }
           set { _x = value; }
        }
}
class ClsWithProperty2
{
        public int Y { get; set; }
        public ClsWithProperty2()  { Y = 0; }

}
static void Main(string[] args)
{
        ClsWithProperty a = new ClsWithProperty();   ClsWithProperty2 b = new ClsWithProperty2();
        a.X = 1; b.Y = -1; // setter
        Console.Write(a.X + b.Y); // getter
}
```

Основные понятия ЯП - данные и операции

Данные группируются в **типы данных**

Каким образом?

Н.Вирт ("Систематическое программирование. Введение"): тип данных характеризуется множество значений (=> перечислимый тип, как простое перечисление множества значений).

Основные понятия ЯП - данные и операции

Данные группируются в **типы данных**

Каким образом?

Н.Вирт ("Систематическое программирование. Введение"): тип данных характеризуется множество значений (=> перечислимый тип, как простое перечисление множества значений).

Современная точка зрения:

ТД = Множество Значений + Множество Операций

МЗ = структура данных

МО = сигнатура типа

Что главное? Сигнатура

АТД = Сигнатура (абстрагируемся от структуры, сосредотачиваемся на операциях)

Основные понятия ЯП - связывание

Связывание - процесс установления связи между элементом программы и конкретным атрибутом или характеристикой.

Сводится к выбору атрибута из (возможно конечного) набора атрибутов.

Время связывания — это момент установления этой связи.

Основные виды времен связывания:

- *Во время выполнения программы - динамическое связывание.*
- *При трансляции (компоновке) - статическое связывание*.

Основные понятия ЯП - связывание

Статическое связывание:

- *По выбору программиста.* Это, например, связывание объекта данных с именем, имени с типом (в языке, подобном Паскалю) и так далее.
- *По выбору транслятора*. Это, например, связывание относительного адреса (не путайте с абсолютным) локальных переменных (в языке, подобном Паскалю или Си). Конкретный вариант такого связывания, как правило, определяется реализацией языка.
- *По выбору компоновщика (загрузчика, редактора связей).* При ссылке на переменную или вызове подпрограммы из другого модуля связать конкретные адреса с местом ссылки или вызова возможно только на этапе компоновки, когда доступны все используемые модули и библиотеки, и известен их относительный порядок.

Основные понятия ЯП - связывание

Динамическое связывание:

- *При входе в блок (тело подпрограммы)*. Например, в этот момент локальные переменные блока и формальные параметры подпрограммы связываются с атрибутом-адресом. Связывание фактических и формальных параметров тоже происходит в этот момент. Такой способ мы будем называть ***квазистатическим***.
- *В произвольной точке программного кода.* К этому виду относится, например, связывание переменной и значения, происходящее во время выполнения оператора присваивания. Другие примеры: связывание адреса и объекта данных при динамическом распределении памяти, выбор обработчика исключительной ситуации при распространении исключения

Основные понятия ЯП - связывание

Другие виды времени связывания:

- *во время реализации языка* (например, выбор максимального и минимального значения типа int при реализации транслятора с языка С)
- *во время определения языка* (например, номенклатуру типов данных в языке).

Основные понятия ЯП - связывание

Важность понятия связывания.

Разберем пример (на первый взгляд тривиальный).

Что такое константа и чем она отличается от переменной?

(А какая ПП?)

Изменяемое - неизменяемое (?) - когда, как изменяемое?

Основные понятия ЯП - связывание

Важность понятия связывания.

Что такое константа и чем она отличается от переменной?

(А какая ПП?)

Нужно понять - какие связывания и время связывания.

Главное связывание в императивной парадигме:

Объект данных <=> значение

Основные понятия ЯП - связывание

Важность понятия связывания.

Что такое константа и чем она отличается от переменной?

(А какая ПП?)

Нужно понять - какие связывания и время связывания.

Главное связывание в императивной парадигме:

Объект данных <=> значение

Отличие var от const - во времени связывания

Статическое (const) versus динамическое (var)

Основные понятия ЯП - связывание

Важность понятия связывания.

Что такое константа и чем она отличается от переменной?

(А какая ПП?)

Нужно понять - какие связывания и время связывания.

Главное связывание в императивной парадигме:

Объект данных <=> значение

Отличие var от const - во времени связывания

Статическое (const) versus динамическое (var)

Но тут возникает главный вопрос - а когда именно связывается значение с константой?

Основные понятия ЯП - связывание

Когда именно связывается значение с константой?

При компиляции (чисто статически) - Паскаль, C++, C#....

Компилятор "знает" и может использовать значение константы везде

Паскаль:

```
const N = 1024;
var arr: array [0..N-1] of integer;
```

C++:

```
const int N = 1024;
int arr[N];
```

C#: пример с массивами не пройдет, т.к. массивы - ДИНАМИЧЕСКИЕ объекты (нет статического связывания массива с адресами в памяти - только в динамической памяти).

```
const string message = "This is a static constant";
```

А вот что там в C?

Основные понятия ЯП - связывание

Когда именно связывается значение с константой?

А вот что там в C?

K&R C:

#define N 1024

C90:

const int N = 1024;

Основные понятия ЯП - связывание

Когда именно связывается значение с константой?

А вот что там в С?

K&R C:

```
#define N 1024
#define A 10.0
#define H ((A)/((N) + 1))
```

Ну зачем эти скобки???

Основные понятия ЯП - связывание

Когда именно связывается значение с константой?
А вот что там в С?
K&R C:
#define N 1024
#define A 10.0
#define H ((A)/((N) + 1))
C90 (aka ANSI C89):
const int N = 1024;
const double A = 10.0;
const double H = A/(N+1);
Ну совсем цивильно!!!

Основные понятия ЯП - связывание

Когда именно связывается значение с константой?

А вот что там в C?

K&R C:

```
#define N 1024
```

C90(aka ANSI C89):

```
const int N = 1024;
int arr[N];  /* ошибка компиляции!!! */
```

Основные понятия ЯП - связывание

Когда именно связывается значение с константой?
А вот что там в C?
K&R C:
#define N 1024
C90(aka ANSI C89):
const int N = 1024;
int arr[N]; /* ошибка компиляции!!! */
C++ (всегда!!!)
const int N = 1024;
int arr[N]; // как в Паскале...

Основные понятия ЯП - связывание

Когда именно связывается значение с константой?

А если понятие константы определяется не временем связывания, а именно НЕИЗМЕННОСТЬЮ ее значения после инициализации?

Тогда возникают два понятия константы:

- статическая инициализация
- динамическая инициализация

Основные понятия ЯП - связывание

Когда именно связывается значение с константой?

А если понятие константы определяется не временем связывания, а именно НЕИЗМЕНЯЕМОСТЬЮ ее значения после инициализации?

Тогда возникают два понятия константы:

- статическая инициализация
- динамическая инициализация

C#:

```
 class ConstSample
{
    public const int N = 1024; // статическая инициализация и только статический член данных
    public readonly double H; // динамическая инициализация
    public ConstSample(double Dist)
    {
        H = Dist / (N + 1);
    }
}
```

Основные понятия ЯП - связывание

Когда именно связывается значение с константой?

А если понятие константы определяется не временем связывания, а именно НЕИЗМЕНЯЕМОСТЬЮ ее значения после инициализации?

Тогда возникают два понятия константы:

- статическая инициализация
- динамическая инициализация

C++:

```
void foo(const int bar); // время связывания bar???
class ConstSample
{
public:
    static const int N = 1024; // угадайте год стандарта!!!
    const double H; // динамическая инициализация, квазистатическое связывание (как в C#)
    ConstSample(double dist) : H (dist / (N + 1))  {  }
    void bar(int i) {
        foo (j+ 25);
    }
};
```

Основные понятия ЯП - связывание

Другой пример важности понятия (времени) связывания - виртуальные и невиртуальные функции языка C++.

Отличие - только во времени связывания **ВЫЗОВА** функции.

Реализация (сгенерированный код) виртуальной и невиртуальной функции не отличаются.

```
class X {
public:
   void g() { cout << "X::g()"; }
   void f()  { g(); }
   virtual void vf() { g();}
};
class Y : public X {
public:
   void g() { cout << "Y::g()"; }
   void f()  { g(); }
   virtual void vf() { g(); }
};
```

Основные понятия ЯП - связывание

Другой пример важности понятия (времени) связывания - виртуальные и невиртуальные функции языка C++.

Отличие - только во времени связывания **ВЫЗОВА** функции.

Реализация (сгенерированный код) виртуальной и невиртуальной функции не отличаются.

```
class X {
  void g() { cout << "X::g() "; }
public:
   void f()  { g(); }
   virtual void vf() { g();}
};
void sample(X& x) { // в этой точке транслятор НИЧЕГО не знает о наличии преемников, класса Y и т.д.
   x.f(); x.vf();
}
//    может быть в в другом модуле
class Y : public X {
   void g() { cout << "Y::g() "; }
public:
   void f()  { g(); }
   virtual void vf() { g(); }
};
void foo() {
   X ax; Y ay;
   sample (ax); // разница не видна: X::g() X::g()
   sample (ay); // разница видна : X::g() Y::g()
}
```

Основные понятия ЯП - связывание

Другой пример важности понятия (времени) связывания - виртуальные и невиртуальные функции языка C++.

Отличие - только во времени связывания **ВЫЗОВА** функции.

Реализация (сгенерированный код) виртуальной и невиртуальной функции не отличаются.

Снятие виртуальности вызова:

```
void bar(X& x) {
   x.f(); x.X::vf();
}
void foo() {
    X ax; Y ay;
    sample (ax); // разница не видна: X::g() X::g()
    sample (ay); // разница видна : X::g() Y::g()
    bar(ax);  // как sample (ax)
    bar(ay); // нет разницы - как bar(ax);
}
```

Основные понятия ЯП - связывание

Компилируемость, интерпретируемость и время связывания.

Языки бывают (из литературы по ЯП):

компилируемые, интерпретируемые, типизированные, бестиповые (?), слабо типизированные, сильно типизированные, строго типизированные, динамические, статические, bla bla....

О чем идет речь?

Основные понятия ЯП - связывание

Компилируемость, интерпретируемость и время связывания.

Языки бывают (из литературы по ЯП):

компилируемые, интерпретируемые, типизированные, бестиповые (?), слабо типизированные, сильно типизированные, строго типизированные, динамические, статические, bla bla....

О чем идет речь?

На самом деле:

любой ЯП может компилироваться

любой ЯП может интерпретироваться

в любом языке есть данные => есть типы данных

Вопрос об одном из ключевых (или даже самом ключевом) понятий связывания

Основные понятия ЯП - связывание

Связывание:

объект данных <=> тип данных

Статическое связывание => статический (=статически типизированный язык)

Динамическое связывание => динамический (=динамически типизированный язык, он же "бестиповый" язык - "бесстатическитиповый").

Термины "слабо-сильно-строго"-типизированный язык весьма размыты и неопределенны.

"Строго" - не будем использовать в этом курсе.

Слабо-сильно.

Тип (чего?) может меняться или не может меняться.

На самом деле:

Тип объекта, который инициализирован и размещен в памяти, не может изменяться.

Может изменяться тип ссылки на объект.

Слабо типизированный язык - тип ссылки может изменяться (динамически) при выполнении операций над ссылкой.

Сильно типизированный язык - тип ссылки не может изменяться (динамически) при выполнении операций над ссылкой.

Основные понятия ЯП - связывание

Компилируемые ЯП - компиляция должна давать существенный выигрыш по сравнению с компиляцией.

Интерпретируемые ЯП - компиляция не дает существенного выигрыша.

Определяется временем связывания типа с данными => возможность выбрать оптимальную реализацию статически.

Пример - Lisp:

(eval (read))

Пример - Java и JavaScript:

string s1,s2; ..... s1 + s2 // только операция конкатенации строк

var s1, s2; ..... s1 + s2 // любая допустимая + операция в JavaScript (строки, числа) - выбор в run-time