Прошлые лекции

Определение ЯП - инструмент планирования поведения исполнителя

Более узко - формальная нотация для записи компьютерных программ

Только индустриальные ЯП

ЯП и ПП

Императивная

Функциональная

Логическая

Господствует сейчас - объектно-императивная

Язык С

С - 1969

Линия языков - CPL=>BCPL=>B=>C

(от B - скобки, синтаксис управляющих конструкций, устаревший синтаксис функций и auto....)

B + система типов (вместо слов, байтов и байтовых строк) => C

Потеря - //

Простой и отвечающий своей главной цели

Высокоуровневый и машинно-независимый ЯА - пока вне конкуренции в своей нише

K&R C (1978)

Стандарты: C89 (ISO-C90), C99, C11, C18, следующий (C2x -не ранее 2021)

Язык Go (Golang)

2007-2009 гг (Google - К.Томпсон, Р.Пайк, Р.Гризмер)

Отказ от некоторых концепций ООЯП (понятие класса с наследованием реализации, перегрузка функций, обобщенное программирование)

Отказ от большинства операций с побочным эффектом и адресной арифметики (i++ - это оператор, а не выражение!)

Автоматическая сборка мусора

Очень развитая многопоточность

Есть элементы ФП (замыкания, анонимные функции...)

Простота, стабильность, поддержка

Ниша - разработка сервисов и утилит

Но не системное программирование!!! (Опыт ОС Fuchsia - Fuchsia Platform Source Tree)

Императивная парадигма (язык Python)

Ведущий язык в нише "научного" программирования, где нет огромного количества вычислений с плав. точкой (number crunching)

Интерпретируемый язык с динамической типизацией и автоматической сборкой мусора, наследник языка ABC (голландская замена Basic и Паскаль - для обучения, прототипирования и т.п.)

Первая версия - 1991. Гвидо Ван Россум (BDFL до 2018)

Python 2 - 2000..... Python 2.7.18 - официальное развитие закончено в 2020 (вместо 2015)

Python 3 (не полностью совместим с 2) - 2008 ..... 3.8.5 (но может уже и больше....)

https://www.python.org/dev/peps/pep-0020/

Вообще развитие языка - сообщество питонистов - роль "стандартов" играют PEP (Python Enhancement Proposals)

Никаких ANSI-ISO стандартов!

Развитие языка - есть существенные элементы ФП (анонимные функции, замыкания, ленивые генераторы...), объекты, классы, наследование, сопрограммы. Очень богатая стандартная библиотека + огромный набор свободных библиотек.

Почему стал индустриальным?

- чисто научные библиотеки и модели становятся частью промышленных разработок
- веб--программирование

Императивная парадигма (язык Python)

Ввод - sys.stdin.read() => весь поток в строку

Реверса строк нет (и это правильно!).

Но: есть очень гибкое понятие вырезки (из массива-списка, строки....)

s = "0123456"

s[0:3] <=> "012"

s[:3] <=> "012"

s[4:] <=> "456"

s[:] <=> "0123456" - полная копия строки

Императивная парадигма (язык Python)

И есть шаг (по умолчанию 1):

s = "0123456"

s[0:5:2] <=> "024"

Фокус: шаг может быть отрицательным

s[3:0:-1] <=> "321"

s[3::-1] <=> "3210"

Императивная парадигма (язык Python)

И есть шаг (по умолчанию 1):

s = "0123456"

s[0:5:2] <=> "024"

Фокус: шаг может быть отрицательным

s[3:0:-1] <=> "321"

s[3::-1] <=> "3210"

ну и наконец:

s[::-1] <=> "6543210"

Императивная парадигма (язык Python)

Шаг может быть отрицательным

s[::-1] <=> "6543210"

А вот и программа (полная!):

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

Рекорд, который трудно побить....

Объектно-императивная парадигма

Объект <=> состояние + поведение

Состояние => структура данных (C++ - члены-данные, Smalltalk - переменные)

Поведение => процедуры и функции, аргументом которых является объект (C++ - члены-функции, Smalltalk - методы доступа)

Объекты посылают друг другу сообщения => посылка сообщения - вызов соответствующего метода класса (C++ - виртуальные методы, Smalltalk - динамический поиск метода-обработчика сообщения).

Объектно-императивная парадигма

ООП в индустриальных ЯП - подчиненная парадигма

Понятие хранимого и изменяемого состояния - главное свойство императивности

Объект <=> состояние + поведение

Объекты посылают друг другу сообщения. При обработке сообщений объект может менять свое состояние и посылать сообщения, меняя тем самым состояние других объектов.

А как это выглядит с точки зрения императивного ЯП?

Объектно-императивная парадигма

Объекты с идентичной структурой состояния и идентичным поведением объединяются в понятие типа данных.

Классы (C++ и далее) - после SmallTalk - главное понятие ООП

Модули (Оберон, Ада)

Важный момент - отношения между классами (вспомним 2 курс).

- наследование
- включение
- референция (вместо подобъекта - ссылка на него)
- и т.д.

Объектно-императивная парадигма

Объекты с идентичной структурой состояния и идентичным поведением объединяются в понятие типа данных.

Классы (C++ и далее) - после SmallTalk - главное понятие ООП

Модули (Оберон, Ада)

Важный момент - отношения между классами (вспомним 2 курс).

- наследование
- включение
- референция (вместо подобъекта - ссылка на него)
- и т.д.

Наша задача - слишком мала для проявления потенциала ООЯП

НО: важно посмотреть на отличия от чисто императивной парадигмы

Объектно-императивная парадигма (язык C#)

```
using System;
public class Program
{
    static void Main(string[] args)
    {
        string s = Console.In.ReadToEnd();
        char[] seq = s.ToCharArray();
        Array.Reverse(seq);
        Console.Write(seq);
    }
}
```

Объектно-императивная парадигма

Почему не Java?

Задача ОЧЕНЬ простая

https://stackoverflow.com/questions/309424/how-do-i-read-convert-an-inputstream-into-a-string-in-java

2018 (Java 11):

Java 11 – Files readString() API

String content = Files.readString(filePath);

А просто с потоком нельзя?

Обобщенное программирование (язык C++)

В рамках ООП

Статическая параметризация типов - механизм шаблонов

Стандартная библиотека C++ = C RTL + STL(набор шаблонов)

STL = контейнеры + итераторы + алгоритмы

Для данной задачи - какой алгоритм и какой контейнер?

cin => контейнер => либо реверс контейнера, либо вывод в cout обратном порядке

Обобщенное программирование (язык C++)

В рамках ООП

Статическая параметризация типов - механизм шаблонов

Стандартная библиотека C++ = C RTL + STL(набор шаблонов)

STL = контейнеры + итераторы + алгоритмы

Для данной задачи - какой алгоритм и какой контейнер?

cin => контейнер => либо реверс контейнера, либо вывод в cout обратном порядке

Какой контейнер? - любой с двунаправленными итераторами (вектор или список)

Обобщенное программирование (язык C++)

В рамках ООП

Статическая параметризация типов - механизм шаблонов

Стандартная библиотека C++ = C RTL + STL(набор шаблонов)

STL = контейнеры + итераторы + алгоритмы

Для данной задачи - какой алгоритм и какой контейнер?

cin => контейнер => либо реверс контейнера, либо вывод в cout обратном порядке

Какой контейнер? - любой с двунаправленными итераторами (вектор или список)

А алгоритмы причем?

Обобщенное программирование (язык C++)

В рамках ООП

Статическая параметризация типов - механизм шаблонов

Стандартная библиотека C++ = C RTL + STL(набор шаблонов)

STL = контейнеры + итераторы + алгоритмы

Для данной задачи - какой алгоритм и какой контейнер?

cin => контейнер => либо реверс контейнера, либо вывод в cout обратном порядке

Какой контейнер? - любой с двунаправленными итераторами (вектор или список)

А алгоритмы причем?

copy(begIt, endIt, whereIt)

Потоки - тоже контейнеры!

Обобщенное программирование (язык C++)

Потоки - тоже контейнеры! И у них есть итераторы

Два копирования:

std::vector<char> v;

copy_from_cin_to_v

copy_from_v_to cout_in_reverse_order

Обобщенное программирование (язык C++)

std::vector<char> v;

copy_from_cin_to_v: copy(begin_it_cin, end_it_cin, iterToV)

1) диапазон итератора для потока ввода?
2) Как вставлять в массив с помощью итераторов?

Обобщенное программирование (язык C++)

std::vector<char> v;

copy_from_cin_to_v: copy(begin_it_cin, end_it_cin, iterToV)

диапазон итератора для потока ввода?

Специальный класс для итераторов потоков ввода:

istream_iterator<T> - здесь T - тип лексем, которые образуют поток ввода (литеры, числа, строки....)

Обобщенное программирование (язык C++)

std::vector v;

copy_from_cin_to_v: copy(begin_it_cin, end_it_cin, iterToV)

диапазон итератора для потока ввода?

Специальный класс для итераторов потоков ввода:

istream_iterator<T> - здесь T - тип лексем, которые образуют поток ввода (литеры, числа, строки....)

Два конструктора:

istream_iterator<T>(in_stream) - аналог begin()

istream_iterator<T>() - аналог end()

Обобщенное программирование (язык C++)

vector<char> v;

copy_from_cin_to_v: copy(begin_it_cin, end_it_cin, iterToV)

диапазон итератора для потока ввода?

Специальный класс для итераторов потоков ввода:

istream_iterator<T> - здесь T - тип лексем, которые образуют поток ввода (литеры, числа, строки....)

Два конструктора:

istream_iterator<T>(in_stream) - аналог begin()

istream_iterator<T>() - аналог end()

Диапазон: [istream_iterator<char>(cin), istream_iterator<char>()]

Обобщенное программирование (язык C++)

vector<char> v;

copy_from_cin_to_v: copy(begin_it_cin, end_it_cin, iterToV)

Как вставлять в массив с помощью итераторов?

Снова специальный класс итератора

back_inserter(container) =>

back_inserter(v)

Обобщенное программирование (язык C++)

```
vector<char> v;
copy_from_cin_to_v: copy(begin_it_cin, end_it_cin, iterToV)
copy(istream_iterator<char>(cin),
        istream_iterator<char>(),
         back_inserter(v)
)
```

Обобщенное программирование (язык C++)

```
std::vector<char> v;
std::copy(istream_iterator<char>(cin),
        istream_iterator<char>(),
         back_inserter(v)
);
```

copy_from_v_to cout_in_reverse_order

Обратное копирование (из v в cout) - по аналогии:

```
std::copy(v.rbegin(), v.rend(), ostream_iterator<char>(cout));
```

Обобщенное программирование (язык C++)

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;

int main() {
	vector<char> cont;
	copy(istream_iterator<char>(cin), istream_iterator<char>(), back_inserter(cont));
	copy(cont.rbegin(), cont.rend(), ostream_iterator<char>(cout));
	return 0;
}
```

Обобщенное программирование (язык C++)

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;
// а если последовательность нужна далее, то можно и перевернуть ее "in situ"
int main() {
	vector<char> cont;
	copy(istream_iterator<char>(cin), istream_iterator<char>(), back_inserter(cont));
	reverse(cont.begin(), cont.end());
	copy(cont.begin(), cont.end(), ostream_iterator<char>(cout));
	return 0;
}
```

ЯП и парадигмы программирования

Даже для одной парадигмы - какой ЯП "лучше"?

Эффективность, быстрота разработки, надежность...

Много противоречивых критериев оценки => нет универсального языка, хотя попытки его создать - были

ЯП и парадигмы программирования

Даже для одной парадигмы - какой ЯП "лучше"?

Эффективность, быстрота разработки, надежность...

Много противоречивых критериев оценки => нет универсального языка, хотя попытки его создать - были

1964 - PL/I ("язык-оболочка")

1968 - Алгол-68("язык-ядро")

1979-1980 - Ада (83->95->2007->2012- ISO)

Первый язык, разработанный по заранее сформулированным критериям и с соревновательной процедурой

С начала 90-х действовало правило использования (в DoD) только программ на языке Ада и с использованием сертифицированных компиляторов

Часто не соблюдалось и отменено в 2000-х

"Идеальный" универсальный ЯП

Чем дальше развиваются технологии и расширяется область разработки ПО, тем больше ясности, что "наилучшего" ЯП не существует для сколь-нибудь обширной проблемной области

Java (1995-2005-....2018....) - задуман как универсальный общий язык для сети Интернет (WORA - **W**rite **O**nce **R**an **A**nywhere)

И что?

JavaScript - defacto - стандарт для клиентской части веб-приложений (альтернатива VBS)

Реальная альтернатива - не один язык, а Web Assembly (LLVM) с поддержкой теоретически ЛЮБЫХ ЯП

ЯП и парадигмы программирования

Сейчас в индустрии господство объектно-императивной парадигмы

Есть отдельные компании, использующие ФП (Haskell, Scala, Closure), но они (пока?) не делают погоды.

НО: знакомство с концепциями и понятиями "нетрадиционных" парадигм необходимо для развития проф.программиста

Функциональная парадигма

- основные операции - вызов (вычисление) функции и композиция функций
- функции - базисные + определяемые программистом
- функции - объекты первого порядка, то есть могут быть значениями переменных, возвращаемыми и передаваемыми значениями, могут быть созданы динамически (не путать с вызовом функции)
- переменные не меняют значения, а "отождествляются" со своими значениями на все время существования переменной

Таким образом, понятие явного хранимого состояния отсутствует

Функциональная парадигма

Языки:

FP - модельный язык (1978 - Дж.Бэкус - **Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs**)

https://dl.acm.org/doi/10.1145/359576.359579

ML, Hope, Miranda, Haskell

Lisp (Common Lisp) - не является "чистым" ФЯП, там присутствуют различные парадигмы, в. т.ч. императивное:

SETQ V 5

"Настоящее" присваивание переменной V значения 5

Есть и циклы, и последовательность операторов и другие чисто императивные конструкции

CLOS - Common Lisp With Object System - настоящий ООЯП

Почему Lisp?

- первый ЯП с характерными функциональными конструкциями
- модельное подмножество ("pure Lisp") - чистый ФП
- Чистый Лисп - очень компактный и "красивый" язык

Язык Лисп

List **P**rocessing - 1958 - Джон Маккарти (MIT) - вначале чисто теоретический инструмент . Далее (1959-62) превратился в ЯП для обработки символьной информации (текста). Стал de-facto стандартом в нише исследований по ИИ.

Сейчас - фактически семейство языков-диалектов (MuLisp, FranzLisp, InterLisp, EmacsLisp, AutoLisp,....). Самые известные (?) - Scheme (основа курса 6.001 - SICP - в MIT до 2008), Common Lisp - стандарт ANSI 1994.

Почему сейчас будем рассматривать программу на чистом Лиспе?

Схема рассмотрения языков программирования

- Базис
  - скалярный (примитивные ТД и операции)
  - структурный (составные ТД, операторы ...)
- Средства развития (создания новых абстракций) (подпрограммы, модули, средства создания новых ТД и операций)
- Средства защиты (в широком смысле) (механизмы инкапсуляции, межмодульные связи...)

Особенность чистого Лиспа (по сравнению с большинством других ЯП) - простота базиса и средств развития

Функциональная парадигма (язык Лисп)

Атомы языка Лисп (примит. данные)

Атом - символ или целая константа (произвольного размера)

Примеры:

ident 42 + NIL T

Особый атом - (). Другое обозначение - NIL - название - пустой список

Некоторые символы имеют предопределенный смысл ("зашит" в транслятор) - NIL, +, T, чаще всего - имена и знаки встроенных функций (+, CAR, CDR, CONS)

Скалярный базис - все....

Функциональная парадигма (язык Лисп)

Главная (и единственная!) структура данных - S-выражение.

S-выражение => атом | точечная пара

Точечная пара (голова-хвост) => (S-выражение . S-выражение)

примеры:

(a . b) (nil . abc) (nil . nil) (a . (b. c)) (a . (b. (c . nil)))

А где же список (Lisp vs Sep)?

Функциональная парадигма (язык Лисп)

Главная (и единственная!) структура данных - S-выражение.

S-выражение => атом | точечная пара

Точечная пара (голова-хвост) => (S-выражение . S-выражение)

примеры:

(a . b) (nil . abc) (nil . nil) (a . (b. c)) (a . (b. (c . nil)))

А где же список (Lisp vs Sep)?

Список - частный случай S-выражения.

Список - либо () (он же - nil, он же - пустой список), либо точечная пара, в которой голова - это атом или список, а хвост - список

Список => (голова . хвост)

голова => атом | список

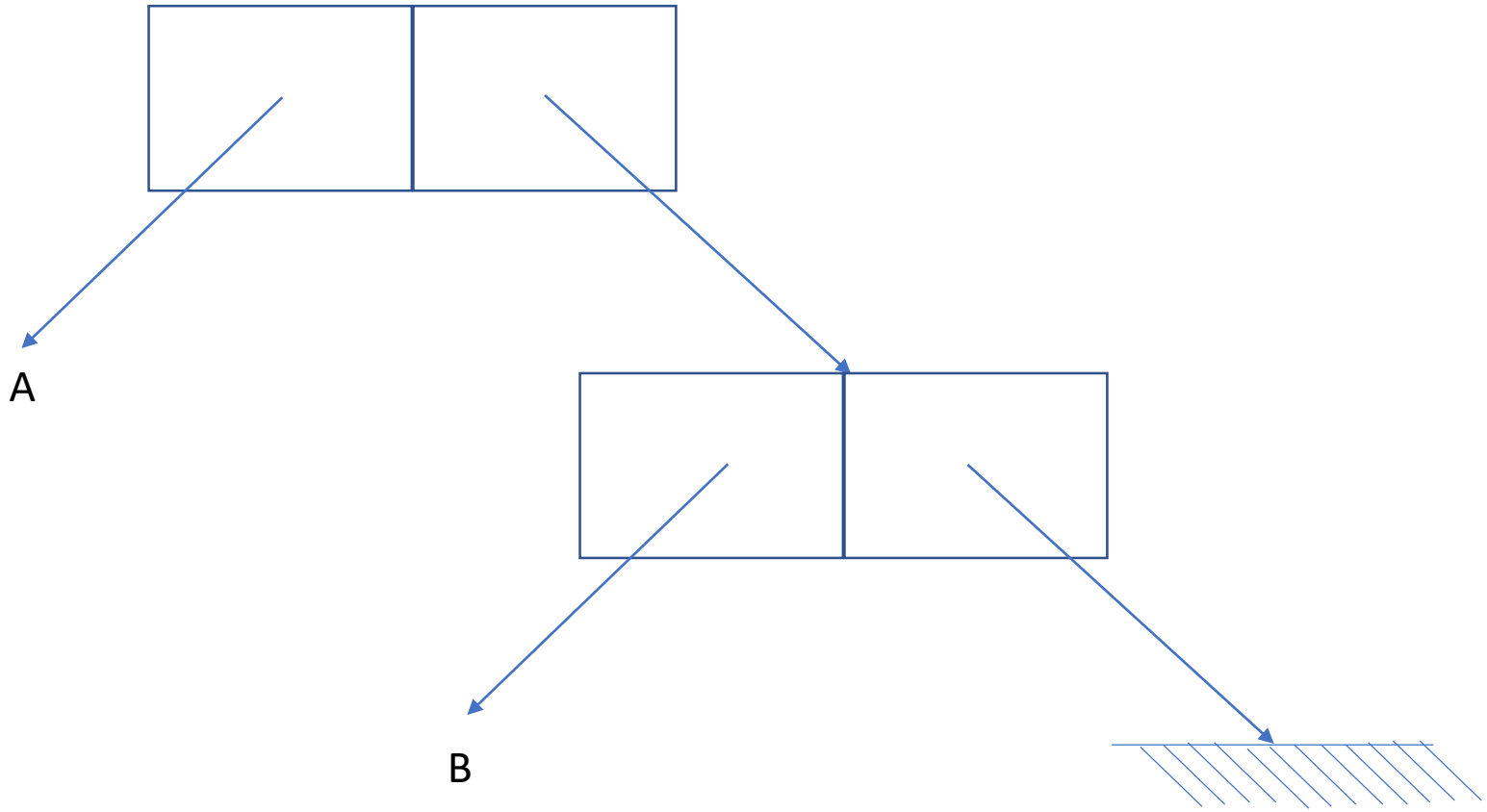
хвост => список

Есть ли список в списке примеров S-выражений?

Функциональная парадигма (язык Лисп)

Графическое изображение списка (или вообще точечной пары)

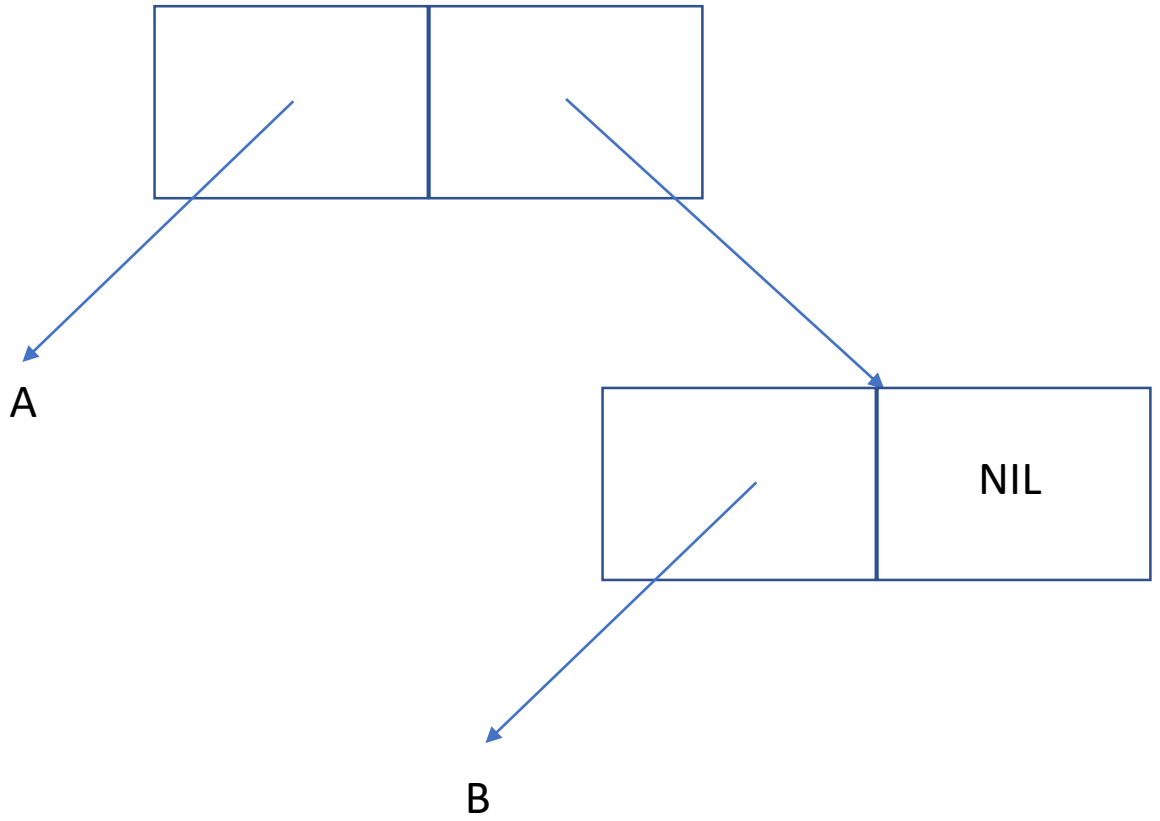
(A.(B.()))



Функциональная парадигма (язык Лисп)

Графическое изображение списка (или вообще точечной пары)

(A.(B.()))



Функциональная парадигма (язык Лисп)

Более компактное изображение списка:

(последовательность листьев через пробел)

(A.(B.NIL) => (A B)

(A . (B. (C . nil))) => (A B C)

((A . nil) . (B. (C . nil))) => ((A) B C)

И т.д.

Базисные ТД рассмотрели все....

Теперь базисные операции (то есть функции, их вычисление и композиция), то есть семантика лисп-вычислителя

Функциональная парадигма (язык Лисп)

Один шаг лисп-вычислителя

Что есть значение? Атом или список

REPL (read - evaluate - print - loop)

Атом-целое => сам в себя

Пустой список => сам в себя

Атом-символ => в свое значение

А откуда значения? Либо в результате функции-присваивания (SET, SETQ) - но это уже просто Лисп, а не "чистый" Лисп, либо при инициализации символа. Где?

Функциональная парадигма (язык Лисп)

Один шаг лисп-вычислителя

Что есть значение? Атом или список

REPL (read - evaluate - print - loop)

Атом-целое => сам в себя

Пустой список => сам в себя

Атом-символ => в свое значение

А откуда значения? Либо в результате функции-присваивания (SET, SETQ) - но это уже просто Лисп, а не "чистый" Лисп, либо при инициализации символа. Где?

Значение символа-переменной, которая является формальным параметром функции, инициализируется (=отождествляется с) вычисленным значением фактического параметра. Не присваивание, поскольку нет изменения предыдущего значения.

В "чистом" ФП вызов функции с одинаковыми фактическими параметрами всегда должен дать один и тот же результат

Функциональная парадигма (язык Лисп)

Вычисление списка (если S-выражение - не список, то ошибка)

(A B C .....)

A - должен быть символом с функциональным значением. Откуда? Либо базисная функция (+, CAR, CDR....), либо определяемая пользователем (об этом ниже).

Первый символ не вычисляется, а все остальные - вычисляются (вопрос - важен ли порядок?). Вычисленные значения позиционно ОТОЖДЕСТВЛЯЮТСЯ с формальными параметрами функции, далее происходит вычисление тела функции. Пока не задаемся вопросом, как задавать новые функции, а считаем, что все функции - из базиса.

Пример:

(+ 1 2 3)

Результат 6

(A B C)

Ошибка (если с символом A не связано никакое функциональное значение)

Аналогично:

(1 2 3 4 5)

Функциональная парадигма (язык Лисп)

Еще стандартные функции:

(CAR S) - возвращает голову точечной пары списка S. Если S - атом, то ошибка. Вопрос: чем будет возвращаемое значение - атомом или списком?

(CDR S) - возвращает хвост точечной пары списка S. Если S - атом, то ошибка. И тот же вопрос.

Странные имена, но традиция сильнее...

Ну напоследок: (CONS A B) - одна из немногих функций, которая возвращает точечную пару общего вида, A - хвост, B - голова.

Если эта пара не является списком, то это проблемы программиста, что с ним дальше делать....

Прежде, чем приводить примеры, зададимся вопросом - будут ли программы, использующие эти функции алгоритмически полными?

НЕТ! Только бесконечная рекурсия или заведомо конечное множество операций. Слишком жесткое условие - ЗАРАНЕЕ вычислять ВСЕ аргументы.

Функциональная парадигма (язык Лисп)

Некоторые базисные функции являются специальными, то есть не совсем подчиняющимися общему правилу вычисления функций

Каждая спец. функция - инд.правила вычисления аргументов

(QUOTE X) - отменяет вычисление своего аргумента

Примеры:

без QUOTE =>

(CONS A ()) - ошибка, (CAR (A B C)) - тоже, (CDR (A B C))

(CONS (QUOTE A) ())=> (A)

(CAR (QUOTE(A B C))) => A

(CDR (QUOTE(A B C)))=> (B C)

Функциональная парадигма (язык Лисп)

Некоторые базисные функции являются специальными, то есть не совсем подчиняющимися общему правилу вычисления функций

Все это можно записать короче:

(QUOTE X) => 'X

(CONS 'A ())=> (A)

(CAR '(A B C)) => A

(CDR '(A B C))=> (B C)

Функциональная парадигма (язык Лисп)

Другая важная спец. функция

(IF B S1 S2) -> вычисляет B, если результат вычисления - "истина", то вычисляется S1 , если - "ложь", то вычисляется S2. S1 и S2 не могут быть вычислены одновременно.

Что есть "истина"?

Функциональная парадигма (язык Лисп)

Другая важная спец. функция

(IF B S1 S2) -> вычисляет B, если результат вычисления - "истина", то вычисляется S1 , если - "ложь", то вычисляется S2. S1 и S2 не могут быть вычислены одновременно.

Что есть "истина"?

Это то, что не "ложь".

А "ложь" - это (), он же NIL.

Функции-предикаты (они - обычные):

(NULL S) => T , если S не является (), иначе NIL

Функциональная парадигма (язык Лисп)

Решение задачи:

(print (reverse (read)))

Без read:

(print (reverse '(ABC)))

Функциональная парадигма (язык Лисп)

Решение задачи:

(print (reverse (read)))

Без read:

(print (reverse '(ABC)))

(ABC)

То есть read - читает СПИСОК, а не литеры, reverse - тоже работает со СПИСКОМ, а не литерами

(print (reverse '(A B C)))

(C B A)