Позвольте представить: LISP ...

- Почти все о нём слышали
- Почти никто его не видел
- А он вообще существует ?
- Да ! А вот и он !

Знакомство с Common LISP

- Лисперы знают об этой ”проблеме”, и они придумали шутливый логотип LISP-а на эту тему.
- Лисп на самом деле существует, и он никакой не монстр.
  - Мы попробуем показать, чем является, и чем не является лисп на самом деле.

Мифы

- Что вы слышали о LISP-е ?
  - Он очень старый ?
  - Он очень странный ?
  - Он базируется на какой-то там математической теории ?
  - Используется в САПР-ах ?
  - Язык программирования искусственного интеллекта ?
  - Не нормальный, а какой-то ”функциональный” язык ?
  - Внедрён в наши мозги инопланетянами посредством телепатии ?
  - Вывод: ЛУЧШЕ С НИМ НЕ СВЯЗЫВАТЬСЯ !

Неправда !

- Всё это – мифы, по крайней мере частично.
- LISP – это гораздо более обычный язык программирования, чем вы думаете.
- По своей сути лисп очень прост

Попробуем это показать.

```
;; это печатает элементы списка
;; через запятую
(dolist (el the-list)
  (format t "~%~A," el))
```

История

- LISP – достаточно зрелый язык
- изобретён Джоном МакКарти (John McCarthy) в 1958 году в MIT (Массачусетский технологический институт).
- впервые реализован Стивом Расселом на машине IBM 704.
- Первый полный компилятор лиспа, написанный на лиспе, был реализован в 1962 году Тимом Хартом и Миком Левиным в MIT.
- Второй язык программирования высокого уровня в мире! (первым был FORTRAN)

История (продолжение)

- LISP на самом деле не язык программирования.
- Сейчас существует много LISP-подобных языков:
  - SCHEME (стандарт IEE 1990 года)
  - Emacs LISP
  - Common LISP (1984, стандарт ANSI 1990 года)
- Мы будем говорить о языке Common LISP.

Ключевые особенности

- Основан на идее лямбда-исчисления

Идея в том, что функции и их аргументы – это объекты с одинаковыми правами.

- Синтаксис прост и унифицирован: только атомы и списки атомов.
- Динамически типизируемый: переменная может быть всем, чем угодно.
- Управление памятью: автоматический сборщик мусора.
- Поддержка обработки множеств: списки, векторы, отображения
- и интерпретатор, и компилятор одновременно, а значит, интерактивный.

Язык, гибридный во всех отношениях

- Система типов – динамический, строгий
- Императивный язык программирования
- функциональность: лисп – нестрого функциональный язык.
- ООП: объектно-ориентированный, гибридный.
  CLOS – самая изощрённая и мощная объектная система из ОО-языков.
- расширяемость: макросы могут порождать новые конструкции, пользовательские классы могут расширять стандартные типы.

Базовый синтаксис

(Yoda is a jedi-knight) ; это - список

- Это – список
- последовательность непробельных символов – это атом.
  (атомы в данном примере: YODA, IS, A и JEDI-KNIGHT)
- Пробел – это разделитель элементов списка.
- Скобки – это границы списка.
- Комментарии начинаются с ';' и продолжаются до конца строки
- Многострочные комментарии начинаются с #| и кончаются |# , после чего может опять идти текст программы
- Списки могут быть вложенными: (Yoda likes (sabre plants staff))

Атомы – это ...

- символы: Yoda is
- числа: 1 1.25 12.5e91 123/124
- знаки (character): \Y \o \d \a \Space
- строки: ”Yoda is ...”

  - **Вместе атомы и списки называются S-выражениями (S-expression)**

REPL

- Взаимодействие с лиспом – Read-Evaluate-Print Loop (цикл чтения, вычисления и печати)
  - обычно выглядит как коммандная оболочка операционной системы
- Reader – читает S-выражения
- Eval – вычисляет выражения и получает значения
- Writer – печатает значения
- Затем всё повторяется
- проблемы ? Вызывается отладчик с возможными рестартами и инспектором переменных.

Практика: знакомство с REPL.

- Установите какую-нибудь реализацию COMMON LISP.
  - http://clisp.org - CLISP
  - http://sbcl.org - Steel Bank Common Lisp
  - Или найдите что-то ещё на www.cliki.net
- Попробуйте открыть REPL.
- Введите числа, символы, списки.
- Нажимая Enter, получите значения.
- В конце наберите (QUIT) для выхода.

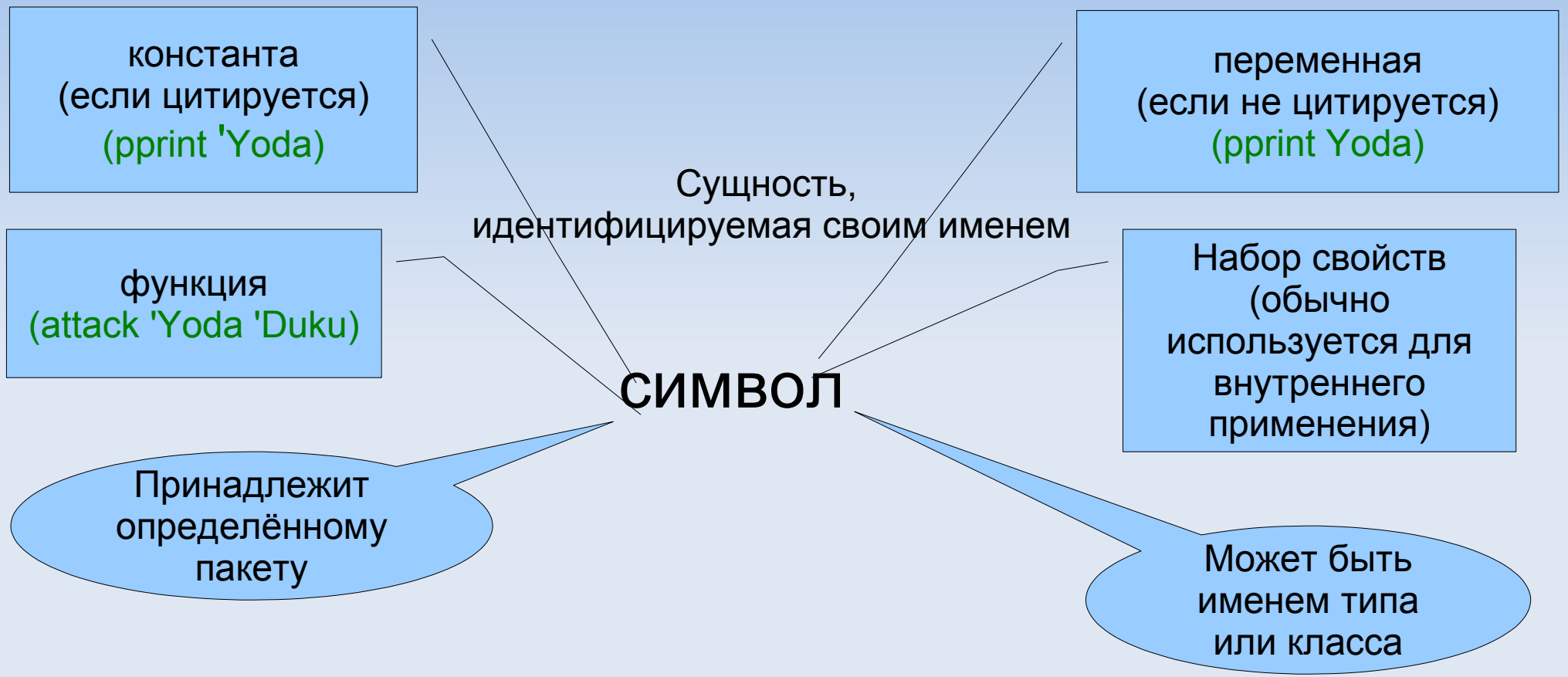
Типы данных

- скалярные
  - символы
  - числа
    - Целые, ”длинные” числа, дроби, числа с плавающей точкой, комплексные.
  - Знаки (character) (однобайтные, многобайтные (unicode), могут включать визуальные атрибуты типа цвета и шрифта)
  - CONS-ы (пара значений)
  - Системные типы (не все)
    - потоки
    - имена файлов
    - пакеты
    - функции
    - метаклассы

... типы данных

- Последовательности (sequience)
  - Списки и деревья
  - Ассоциативные списки (медленные, но лёгкие отображения)
  - векторы (произвольное число измерений; наращиваемые или фиксированного размера)
  - строки (векторы знаков)
  - хэш-таблицы (отображение ключа на значение)
- Сложные структуры (ООП)
  - структуры
  - классы

символы


константа
(если цитируется)
(pprint 'Yoda)
переменная
(если не цитируется)
(pprint Yoda)
Сущность,
идентифицируемая своим именем
функция
(attack 'Yoda 'Duku)
Набор свойств
(обычно
используется для
внутреннего
применения)
СИМВОЛ
Принадлежит
определённому
пакету
Может быть
именем типа
или класса

Пакеты...

- Для устранения конфликтов имён переменных и функций разных приложений и библиотек символы организуются в пакеты.
- Символы из разных пакетов могут иметь одинаковые имена, и при этом они не перепутываются.
- Если нужно найти символ, лисп сначала находит нужный пакет, а потом ищет в нём символ с нужным именем.

… пакеты

Пакеты похожы на каталог книг в библиотеке: сначала ты ищешь раздел, затем в этом разделе – книгу.

- [PACKAGE1]
  - [SYMBOL1]
  - [SYMBOL2]
- [JEDIES]
  - [YODA]
  - [ENIKEN]
  - [LUKE]
- [CREATURES]
  - [YODA]

Имя символа

- Полное имя символа: <пакет>::<символ>
  JEDIES::Yoda
- Если символ экспортируется из пакета: <пакет>:<символ>
  JEDIES:Yoda
- Символ в текущем пакете: <символ>
  (in-package 'Jedies)
  'Yoda
- Особый случай – ключевые слова: имя пакета пустое
  :Yoda

Ключевые слова

- Пакет keywords: имя пакета пустое
  :Yoda
- Все символы внешние, экспортируются автоматом
- Ключевые слова – константы, не могут меняться
- Значение ключевого слова – это само ключевое слово:
  (eq KEYWORDS:Yoda :Yoda ':Yoda)
- Квотирование поэтому не нужно

Практика: знакомство с символами.

- Попробуйте открыть REPL и поработать с символами.
- Введите какой-нибудь символ, например, YODA
  (не забудьте квотировать ! 'YODA)
- Найдите его с помощью функции (find-package)
- Попробуйте другие варианты написания имени символа.
- Попробуйте сделать то же самое с ключевыми словами
- Задайте значение символа: (SETF YODA 42)
- Выведите значение символа
  (квотировать уже не надо: YODA)
- Задайте одному символу значение в виде другого символа.
  Понравилось ?

Элементы построения программ

- переменные
  - лексические
  - динамические
- функции
  - привязанные
  - привязанные локально
  - лямбды (временные функции, определяемые в месте использования)
- классы и структуры

Переменные

- Лексические – определяются в месте их использования
  - 'локальные' (термин не используется в лиспе)

```
(let ((foo ”this is Foo”)
      (bar ”this is bar”)
      (bazz))
  (pprint foo bar bazz))
(pprint foo); Плохо! Уже нет переменной!
```

  - 'глобальные' (термин не используется в лиспе, и на самом деле переменная не глобальная)

```
(defparameter *foo* ”this is Foo”
  ”This is sample global variable
*FOO*”)
```

  - **defvar** почти то же, что и **defparameter**

... Переменные

- Динамические
  - Если переменная не лексическая, и не ”глобальная” или она объявлена как 'special', то она динамическая ('dynamic')
  - ```
    (defparameter *log-file* nil
      ”поток для логирования”)
    ...
    (format *log-file* ”foo”) ; вывод в лог
    ...
    (let ((*log-file* (make-string-output-
    stream))) ;логирование пойдёт в другой поток
      (format *log-file* ”...”)
      (some-func-with-logs))
    ;логирование переключено обратно
    (format *log-file* ”bar”)
    ```
    ```

... переменные

- В Алголо-подобных языках переменные ищутся в последовательно расширяющихся областях видимости: от самого внутреннего блока к внешнему, и до глобальной области видимости.
- В LISP каждый вызов функции или вложенный программный блок создает новое ”пространство имен переменных”, называемое 'frame' (фрейм). Переменные ищутся вверх по последовательности фреймов до самого верхнего уровня – глобальных переменных. При этом вложенность более общая – не только определений, но и вызовов. Лексические переменные ищутся во время компиляции программы. Динамические переменные ищутся во время выполнения программы вверх по стеку вызовов.

Замыкания (closure)

- Функциональное программирование невозможно без замыканий.
- замыкание 'запоминает' переменные из лексического контекста определения функции чтобы потом использовать их во время работы.
- ;; нужно посчитать сумму списка чисел

```
(let ((sum 0))
  (flet ((count-sum (x) (setf sum (+ sum x))))

    ;; сумма будет накапливаться в sum
    (mapcar #'count-sum '(1 2 3 4 5 6 7))
    (pprint sum)))
```

- Но это ещё не всё !

...Замыкания

- ;; нужно посчитать сумму списка чисел
  (let ((sum 0))
    (defun count-sum (x) (setf sum (+ sum x))))

  ;; сумма будет накапливаться в sum

  ;; даже не смотря на то, что sum уже в этом
  ;; блоке нет (мы вышли из LET)
  (mapcar #'count-sum '(1 2 3 4 5 6 7))

  ;; а вот напечатать её уже не получится
  (pprint sum)

...Замыкания

```
;;; так тоже будет работать !
(defvar *sum-zero* nil)
(defvar *sum-func* nil)
(defvar *sum-res* nil)
(let ((sum 0))
  (flet ((zero-sum () (setf sum 0))
         (count-sum (x)(setf sum (+ sum x)))
         (give-sum () "returns sum" sum))
    (setf *sum-zero* #'zero-sum)
    (setf *sum-func* #'count-sum)
    (setf *sum-res* #'give-sum)
))

(funcall *sum-zero*)

;; переменной SUM уже как бы нет, но функция

;; count-sum про неё ещё помнит !
(mapc *sum-func* '(1 2 3 4 5))
(pprint (funcall *sum-res*))
```

Функции

- Синтаксис определения функции
- defun function-name lambda-list [[declaration* | documentation]] form*=> function-name
- lambda-list::= (var*
  [&optional {var | (var [init-form [supplied-p-parameter]])}*]
  [&rest var]
  [&key {var | ({var | (keyword-name var)} [init-form [supplied-p-parameter]])}* [&allow-other-keys]]
  [&aux {var | (var [init-form])}*])
- Рассмотрим подробнее определение функции далее

... функции

- Lambda-list – список формальных параметров функции (все компоненты необязательны)
- Параметры в функции могут передаваться по порядку и по имени.
- Lambda-list может содержать:
  - обязательные параметры, по порядку
  - необязательные параметры, по порядку
  - ключевые параметры, по имени
  - дополнительные параметры, по имени
  - остальные параметры, передаваемые списком

```
(a b c &optional d e &key f g h &rest other)
```

...Функции

- Необязательные и ключевые параметры могут иметь одну из трёх форм:
  - Только параметр:
    ... &optional c d
  - Параметр и его значение по умолчанию:
    ... &key (f 0) (g 'qwert) (h ”never mind”)
  - Параметр, значение по умолчанию, и специальная переменная, говорящая о том, был ли указан этот параметр при вызове функции.
    ... &key (myp 'qwert myp-specified)

... TBC

- See you ...