Lisp Tutorial

Jan Winkler

Institute for Artificial Intelligence
Universität Bremen

Summer Term 2014

Organizational Matters

- Course: Lisp Tutorial, 03-BE-710.98d
- Lecturer: Dipl. Ing. Jan Winkler
  Room: 1.56, TAB, Am Fallturm 1
  28359 Bremen
  Phone: +49 421 218 60019
  E-Mail: winkler@cs.uni-bremen.de
- ... dates (28.04.2014 – ...2014); Monday, 14:00 – 16:00 (**sharp**)
- Credits: 4 ECTS (2 SWS)
- Location: Room 0.31, TAB Building

Topics Covered in this Course

- Introduction to Functional Programming (**Lisp**)
- **ROS** Architecture and **ROSLisp**
- **CRAM** Plan Language and Robot Control Environment
- Construction and Programming of Lego **Mindstorms** Robots
- Groupwork: Robot Construction and Programming
  (group size to be determined)

Topic: Introduction to Lisp

Language Basics:

- Variables and Scopes
- Functions and Methods
- Built-In: Math, List Manipulation

Intermediate Features:

- Common Lisp
- Library: Alexandria

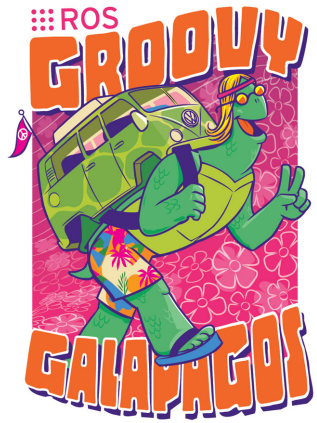
Advanced Features:

- Macros
- File I/O
- Network I/O

Examples on all Topics

Topic: ROS Architecture and ROSLisp

More than just a middleware

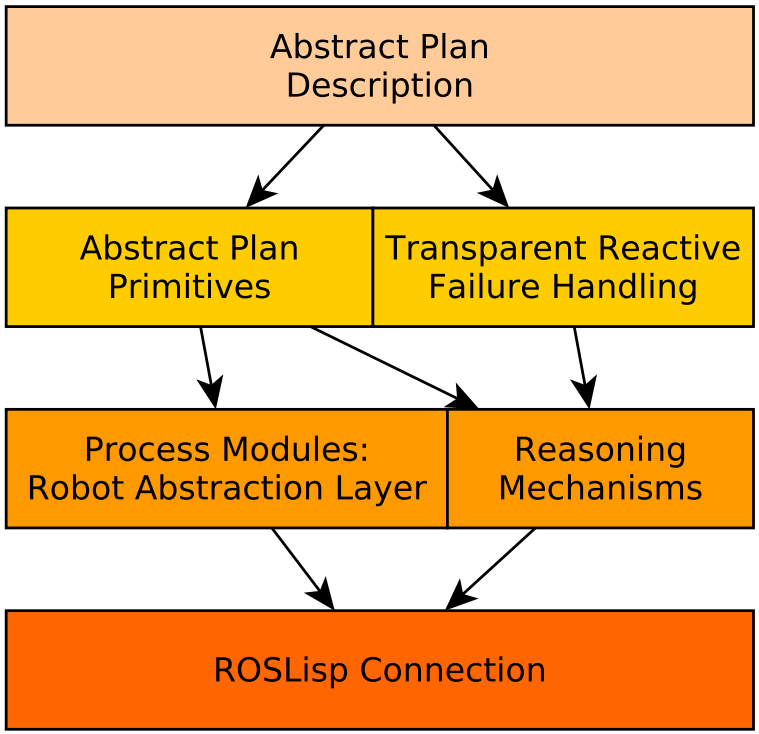


- A "meta" operating system for robots
- A collection of packaging, software building tools
- An architecture for distributed inter-process/inter-machine communication and configuration
- Development tools for system runtime and data analysis
- A language-independent architecture (C++, Python, Lisp, Java, ...)

Topic: CRAM Plan Language for Robot Plans

What is CRAM?

- **Abstraction** Language for Plan-based Control
- **Reactive** Language (with transparent failure handling)
- **Reasoning** machine for situation specific plan parameterization
- Robot independent, **ROS** based
- Written in **Lisp**



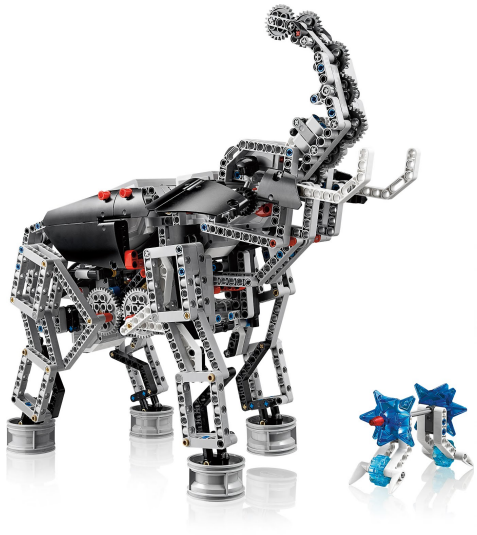
Topic: CRAM Plan Language for Robot Plans

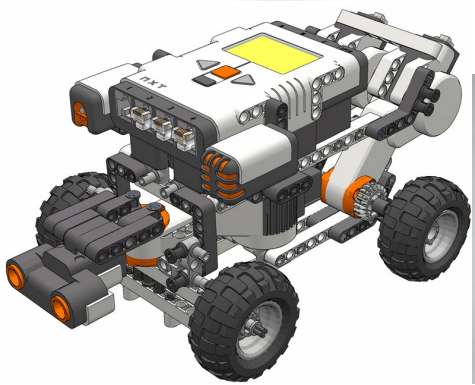
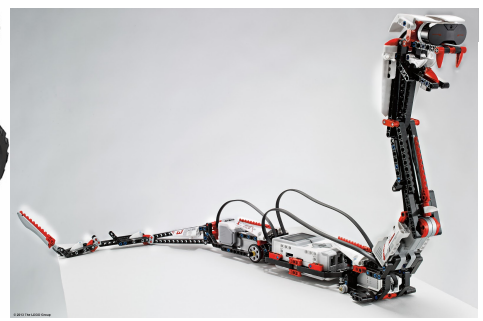
Example

```
(let* ((obj-pose (find-object obj))
       (pre-grasp-pos (calculate-pre-grasp obj-pose))
       (grasp-vector (cl-transforms:make-3d-vector 0 0 -0.1))
       (lift-vector (cl-transforms:make-3d-vector 0 0 0.1)))
  (open-gripper side)
  (take-collision-map)
  (with-failure-handling
      ((no-ik-solution (e)
         (move-to-different-place)
         (retry))
       (link-in-collision (e)
         (setf pre-grasp-pos (new-pre-grasp))
         (retry))
       (trajectory-controller-failed (e)
         (retry)))
    (move-arm-to-point side pre-grasp-pos)))
    ...)
```

Topic: Lego Mindstorms Programming in Lisp

Sample Projects

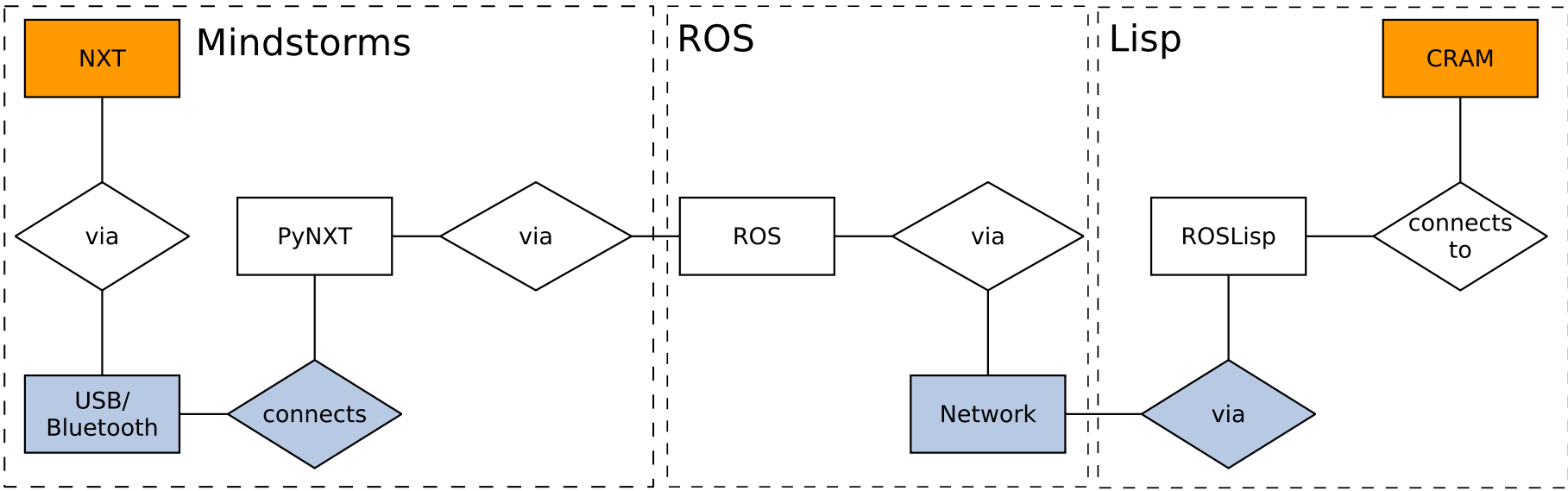




Topic: Lego Mindstorms Programming in Lisp

NXT Brick

Connecting Components

Bringing Topics together



Introduction to Functional Programming

What is Functional Programming?

- Opposing concept to Imperative Programming (C++, Java, etc.)
- Purely Functional: No "State"
- Every method is a function of its parameters
- Not based on side effects

Introduction to Functional Programming

Examples of functional programming languages:

- Haskell
- Clojure
- Scheme
- Emacs Lisp
- Common Lisp ("Grandfather" of Lisp dialects)

Introduction to Functional Programming

We use: SBCL (Steelbank Common Lisp)

- Standardized and widely spread
- Well documented
- Basis for the CRAM plan language

Setup: Emacs, Swank and Slime on Ubuntu 12.04 or 12.10
Virtual Machine:
`http://ai.uni-bremen.de/public/Ubuntu1204LispTutorialCourseVM.ova`

Introduction to Functional Programming

Why Lisp?

- Excellent for exploratory programming (develop code incrementally and interactively): Great for **fast prototyping**
- Ability to extend the language (and syntax) through Macros
- Cross-Platform
- ROS-integration available and stable
- Has object system available: **CLOS** (*Common Lisp Object System*)
- The *programmable programming language*

Introduction to Functional Programming

Live Lisp input:
The REPL (read-eval-print loop)

- Fast Prototyping
- Triggering of larger Functions and Plans

Emacs Cheat Sheet:
GNU Emacs Reference Card
`http://www.ic.unicamp.br/~helio/disciplinas/MC102/Emacs_Reference_Card.pdf`

Lisp Programming Tutorial

Reference Material

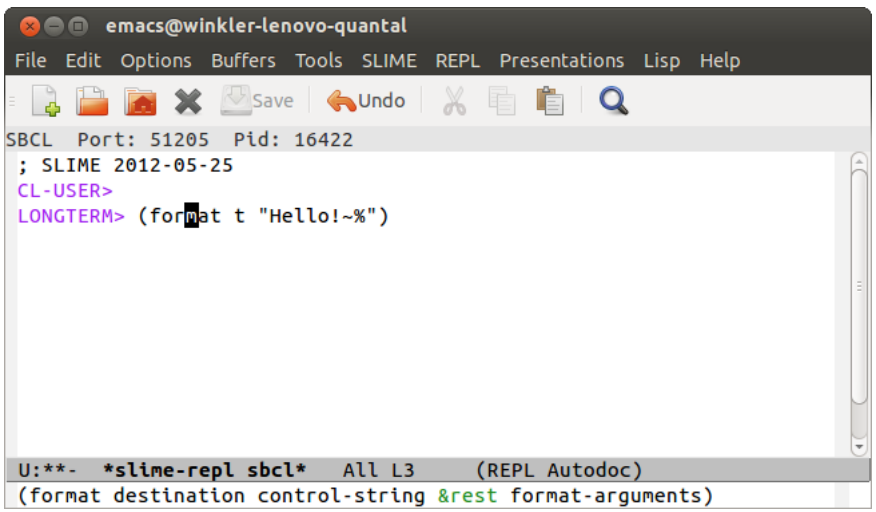
Common Lisp Quick Reference:

`http://www.cheat-sheets.org/saved-copy/clqr-a4-consec.pdf`

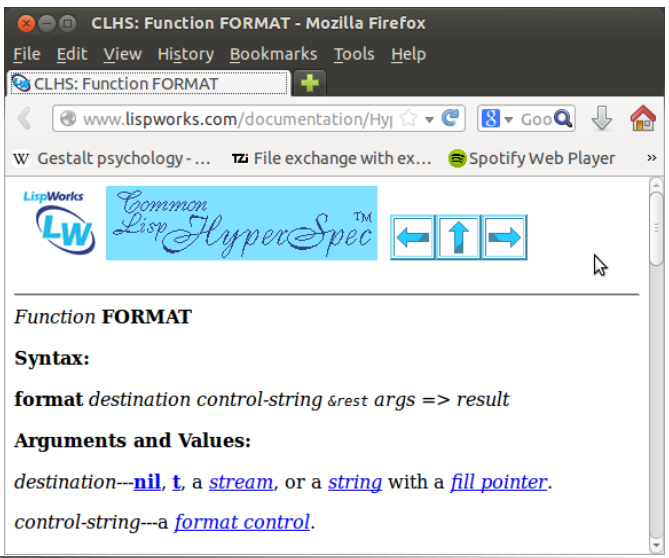
Common Lisp HyperSpec:

`http://www.lispworks.com/documentation/HyperSpec/Front/`

Directly open HyperSpec page for command:



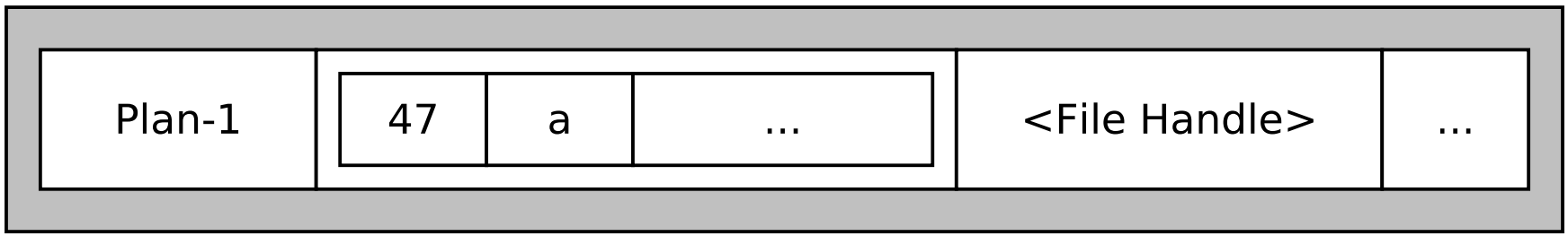
C-x $\Longrightarrow$ C-d h



Lisp Variables and Scopes

Types in Lisp

Basic Structure: **Lists of Lists and Atoms**



Two fundamental Types:

- Atom (Symbol, Object Class Instance, Character, Number)
- List of Atoms

More Types from Classes:

- Arrays, Vectors, Matrices,
- File Handles, Images, ...

Lisp Variables and Scopes

Types in Lisp

Symbol:

- Alphanumeric identifier
- Can have a value (identifier for a variable)
- NOT a string
- Example: `'my-identifier`

Character:

- Character in Unicode range (`www.unicode-table.com`)

Object Class Instance:

- "Pointer" to object
- Based on CLOS
- Can hold arbitrary data, and associates functions (readers, writers, processing)
- Example: `(make-instance 'my-class)`

Numbers:

- Integers, Doubles, ...
- Example: `3`, `10d4`

Lisp Programming Tutorial

Scopes: Where is a variable defined?

Lexical scopes allow for variable definition nesting.

Interlude: The `let` environment

- Defines variable for underlying code segments
- No side-effects (variables disappear when `let` ends)
- Syntax: `(let ((a 5)) ...)`

Lisp Programming Tutorial

Scopes: Where is a variable defined?

A practical example, part 1

```
> (let ((a 5)
        (b 3))
    a)
Result: 5

> (let ((a 5)
        (b 3))
    (let ((a 6)
          (b (+ a 1)))
      a)
Result: 6

> (let ((a 5)
        (b 3))
    (let* ((a 6)
           (b (+ a 1)))
      a)
Result: 6
```

```
> (let ((a 5)
        (b 3))
    b
Result: 3

> (let ((a 5)
        (b 3))
    (let ((a 6)
          (b (+ a 1)))
      b)
Result: 6

> (let ((a 5)
        (b 3))
    (let* ((a 6)
           (b (+ a 1)))
      b)
Result: 7
```

Lisp Programming Tutorial

Scopes: Where is a variable defined?

A practical example, part 2

```
> (let ((a 5)
        (b 3))
    (let* ((a 6)
           (b (+ a 1))))
    a)
Result: 5
```

```
> (let ((a 5)
        (b 3))
    (let* ((a 6)
           (b (+ a 1))))
    b)
Result: 3
```

Nested `lets` only define a sub-scope on the lexical level. When this scope is left, the super-scope is valid again.

Lisp Programming Tutorial

Permanent changes: Side effects

By breaking the functional programming paradigm, side-effects can be enforced:

Side effects when setting variables

```
> (let ((a 6))
    (setf a 7)
    a)
Result: 7
```

```
> (let ((a 6))
    (let ((a 7))
      (setf a 8))
    a)
Result: 6
```

Using `setf`, variables can be set for the current scope.

Lisp Programming Tutorial

Special Role: Global Variables

Global variables (*...*) are always valid on the global scope (but can be overwritten by lexical sub-scopes)

Global Variables

```
> (setf *global-variable* 10)
> (let ((*global-variable* 6))
    *global-variable*)
Result: 6
```

Lisp Programming Tutorial

Special Role: Global Variables

Two common types of global variables:

- `defvar`: Define variables (dynamically changing)
  Syntax: `(defvar var &optional val doc)`
- `defparameter`: Define parameters (fixed)
  Syntax: `(defparameter var val &optional doc)`

Variables and Parameters

```
> (defvar *age* 22 "Age␣of␣the␣person␣in␣question")
> (defparameter *birthdate* "1988-10-02" "The␣birth␣date␣of␣the␣pe
```

Lisp Programming Tutorial

Functions

Functions encapsulate functionality. Examples:

- Math functions: `+`, `-`, `/`, `*`, `sin`, `cos`, ...
  Example: `> (+ 1 2)`
- Output functions: `format`
  Example: `> (format t "Hallo!~%")`

Lisp Programming Tutorial
Functions

Referencing functions: Getting a *reference* on a function for later use

Referencing functions

```
> (let ((func-ref #'+))
    (funcall func-ref 3 4))
Result: 7
```

This can be used when a different function should be selected for different situations in the same mechanism.

Lisp Programming Tutorial

Defining functions

Function definition consists of:

- The keyword `defun`
- The function name (a symbol)
- Its parameters
- Its function body

Example of a function definition

```
> (defun my-function (param-1 param-2)
    (format t "1: ~a ~%2: ~a ~%" param-1 param-2))
> (my-function 'a 'b)
Result: 1: a
2: b
```

Lisp Programming Tutorial

Defining methods

Method definitions are a super-set of function definitions. They consist of:

- The keyword `defmethod`
- The method name (a symbol)
- Its parameters, plus parameter specializers
- Its function body

Example of a method definition

```
> (defmethod my-function (param-1 param-2 &optional param-3)
    (format t "1: ~a~%2: ~a~%" param-1 param-2)
    (when param-3
      (format t "And: ~a~%" param-3)))
> (my-function 'a 'b 'c)
Result: 1: a
2: b
And: c
```

Lisp Programming Tutorial
Defining methods

Some method parameter specializers:

- `optional`: Make this parameter optional
- `key`: Make a key parameter from this parameter
- `rest`: Every parameter after this entry is combined into a list

Example of a method parameter specializers

```
> (defmethod my-function-1 (param-1 &key param-2 (param-3 'a))
    ...)

> (my-function 'a :param-2 'b)
```

When `param-3` is not specified, it holds the symbolic value `a`.

Lisp Programming Tutorial
Defining methods

Using the `rest` parameter specializer:

Example of a `rest` parameters

```
> (defmethod my-function-1 (param-1 &rest parameters)
    ...)

> (my-function 'a 'b 'c 'd)
```

Lisp Programming Tutorial

Function references as parameters

Function references can be given as parameters:

Function references as parameters

```
> (defun my-function-1 (param-function param-value-1 param-value-2)
    (funcall param-function param-value-1 param-value-2))

> (my-function-1 #'+ 1 2)
Result: 3
```

Lisp Programming Tutorial

Anonymous (lambda) functions

Generating functions on the fly using lambda constructs:

- Quick way for defining volatile functions
- Dynamic function generation using parameters

Lambda functions example

```
> (defun make-multiplier (factor deviation)
    (lambda (x)
      (+ deviation (* factor x))))

> (let ((mult (make-multiplier 3 1)))
    (funcall mult 5))
Result: 16
```