Lisp Tutorial

Jan Winkler

Institute for Artificial Intelligence
Universität Bremen

May 5, 2014
Summer Term

Today's Schedule

- Follow-Up: The NXT VM (more details on the VM and the NXT)
- Basic control flow constructs (conditional code execution)
- Libraries:
  - ASDF
  - Common Lisp
  - Alexandria

Follow-Up: The NXT VM

Additional Information

For using the PyNXT tutorial version outside of the Virtual Machine:

- Stripped, Tutorial-oriented version of the PyNXT Repository: `https://github.com/fairlight1337/nxt_python_tutorial_stripped` (no extensive support)
- Add the '`07-lego.rules`' file to your udev rules
- User must be in the '`lego`' group (sign off and back in!)
- Tested on ROS Groovy, Ubuntu 12.04.

During the tutorial course, we will use the VM.

Follow-Up: The NXT VM

Additional Information

Login information for the VM:

- User: `tutorial` (`sudo` rights)
- Password: `tutorial`
- By default, no SSH is installed.

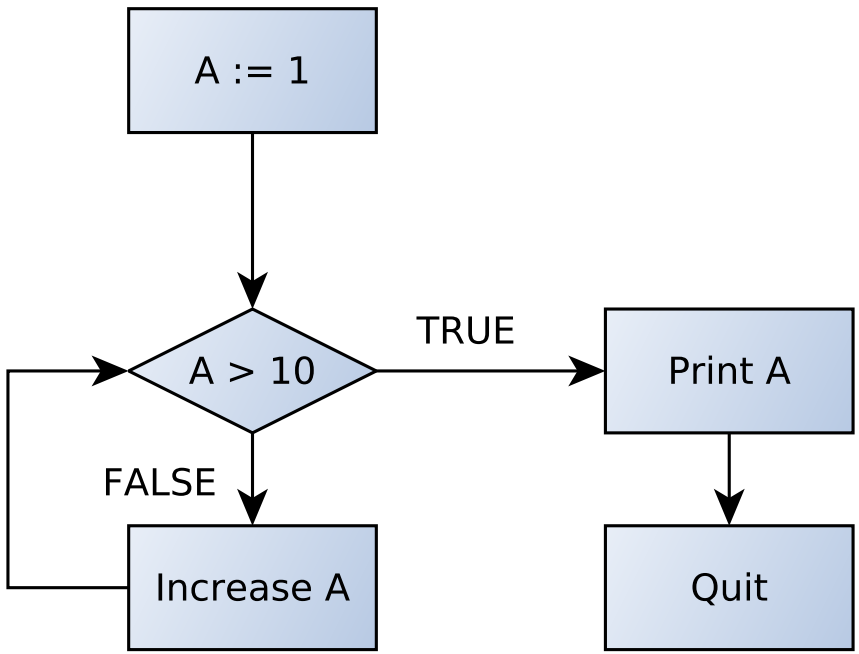
Defined `aliases`:

- `ec`: Starts Emacs in text mode (takes a parameter)
- `repl`: Starts the Lisp REPL in X11 mode, loads slime/swank
- `cm`: Goes into the ROS workspace root folder, performs `catkin_make`, and goes back into the originating folder. Takes two parameters (for example for '`cm --pkg nxt_python`').

Control Flow Constructs

Flow of Execution: `if`

Simple control flow construct: `if`



Control Flow Constructs

Flow of Execution: `if`

Signature:

```
(if test then &optional else)
```

Example of using the if structure

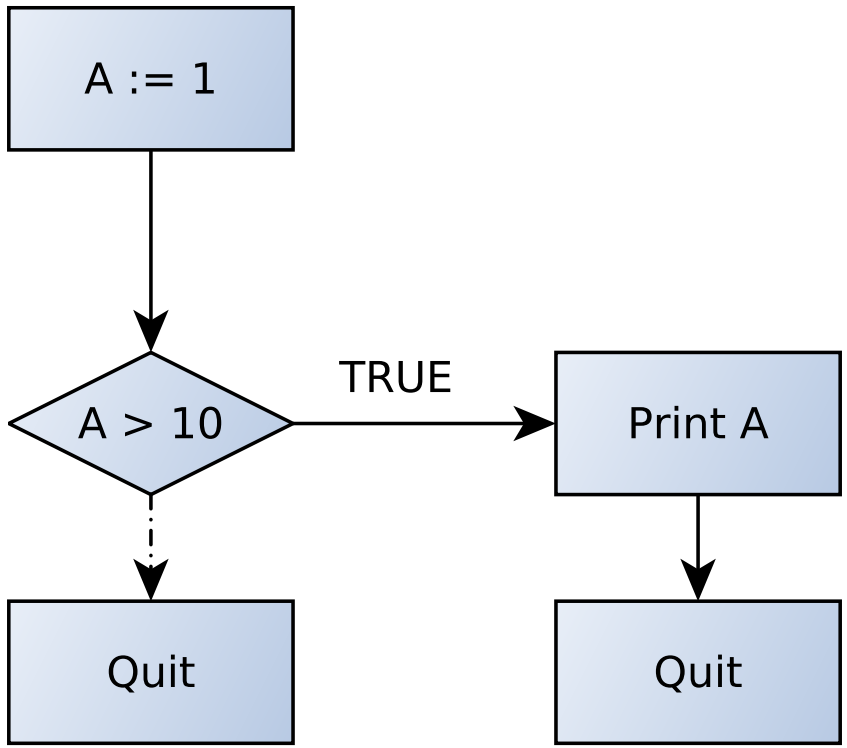
```
> (let ((a 5))
    (if (> a 0)
      (format t "True~%")
      (format t "False~%")))

True
```

http://www.lispworks.com/documentation/HyperSpec/Body/s_if.htm#if

Control Flow Constructs

Flow of Execution: `when`

Simple control flow construct: `when`



Control Flow Constructs

Flow of Execution: `when`

Signature:

```
(when test &body forms)
```

Example of using the when structure

```
> (let ((a -2))
    (when (> a 0)
      (format t "True~%")))

NIL
```

http://www.lispworks.com/documentation/HyperSpec/Body/m_when_.htm#when

Control Flow Constructs

Flow of Execution: `unless`

Opposite of `when`: `unless`

Signature:

```
(unless test &body forms)
```

Example of using the unless structure

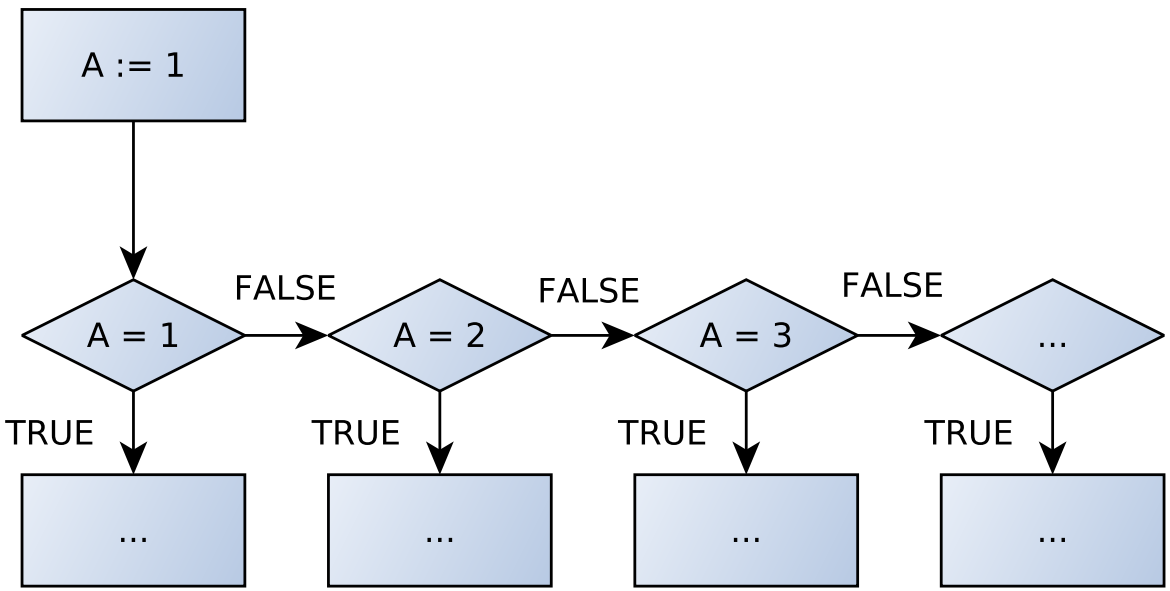
```
> (let ((a -2))
    (unless (> a 0)
      (format t "True~%")))

True
```

http://www.lispworks.com/documentation/HyperSpec/Body/m_when_.htm#unless

Control Flow Constructs

Flow of Execution: `cond`

Simple control flow construct: `cond`



Control Flow Constructs

Flow of Execution: `cond`

Signature:

```
(cond &rest clauses)
```

Example of using the cond structure

```
> (let ((a 2))
    (cond
      ((eql a 1) (format t "A~%"))
      ((eql a 2) (format t "B~%"))
      ((eql a 3) (format t "C~%"))))

B
```

http://www.lispworks.com/documentation/HyperSpec/Body/m_cond.htm#cond

Control Flow Constructs

Flow of Execution: `case`

`cond`'s friend: `case`

Signature:

```
(case keyform &rest cases)
```

Example of using the case structure

```
> (let ((a 2))
    (case a
      (1 (format t "A~%"))
      (2 (format t "B~%"))
      (3 (format t "C~%"))))

B
```

Think "switch/case".

`http://www.lispworks.com/documentation/HyperSpec/Body/m_case_.htm#case`

Control Flow Constructs

Flow of Execution: `and`

Signature:

`(and &rest forms)`

Example of using the and structure

```
> (let ((a 2)
        (b 3))
    (when (and (eql a 2)
               (eql b 3))
      (format t "Fit~%")))

Fit
```

`http://www.lispworks.com/documentation/HyperSpec/Body/m_and.htm`

Control Flow Constructs

Flow of Execution: `or`

Signature:

```
(or &rest forms)
```

Example of using the or structure

```
> (let ((a 2)
        (b 4))
    (when (or (eql a 2)
              (eql b 3))
      (format t "Fit~%")))

Fit
```

`http://www.lispworks.com/documentation/HyperSpec/Body/m_or.htm`

Control Flow Constructs

Flow of Execution: `not`

Signature:

`(not object)`

Example of using the not structure

```
> (let ((a nil))
    (when (not a)
      (format t "a␣is␣empty~%")))

a is empty
```

`http://www.lispworks.com/documentation/HyperSpec/Body/f_not.htm`

Control Flow Constructs

Comparing Values

Different objects require different comparison functions:

- Integers
- Floating point numbers
- Strings
- Objects

Control Flow Constructs

Comparing Values: Integers != Floating point numbers

Compare Int vs. Int or FP vs. FP:

```
(eql x y)
(equal x y)
```

Comparing Int vs. FP doesn't work like this. Coerce first:

Example of using the not structure

```
> (coerse 5 'short-float)
5.0

> (coerse 5 'long-float)
5.0d0
```

Control Flow Constructs

Comparing Values: Integers != Floating point numbers

Simplifier:

```
(= x y)
```

Signature:

```
(= number &rest more-numbers)
```

Explicitly check numbers

Example of using the = structure

```
> (let ((a 5)
        (b 5.0d0))
    (= a b))

T
```

Control Flow Constructs

Comparing Values: Strings

`eql` is for simple values and references. Strings are `lists` of `characters`. Use `string=`:

```
(string= str1 str2)
```

Example of using the string= comparator

```
> (let ((str1 "Test")
        (str2 "Test"))
    (string= str1 str2))

T
```

Control Flow Constructs

Catching errors

Defining handlers for errors; Signature:

```
(handler-bind bindings &body forms)
```

Example of using the handler-bind construct

```
> (handler-bind ((unbound-variable
                  #'(lambda (x) (format t "Unbound: ~a~%" x)))
                 (error
                  #'(lambda (x) (format t "Error: ~a~%" x))))
    unbound-var-name)

Unbound: The variable UNBOUND-VAR-NAME is unbound
Error: The variable UNBOUND-VAR-NAME is unbound
```

`unbound-variable` is a subcondition of `error`. Both are triggered.

Control Flow Constructs

Cleaning up: `unwind-protect`

Definite code execution when failures are signalled:

```
(unwind-protect protected &body cleanup)
```

Mainly for cleanup purposes

Example of using the unwind-protect environment

```
> (unwind-protect
      unbound-var-name
    (format t "Cleanup~%"))

Cleanup
-> ERROR
```

Extension Libraries

Overview

Good Index of Extension Libraries:
`http://common-lisp.net/`

Currently under construction? Check back in a while.

Extension Libraries

Short Interlude: ASDF

ASDF – *Another System Definition Facility*

- `http://common-lisp.net/project/asdf/`
- `http://common-lisp.net/project/asdf-install/tutorial/index-save.html`

It ...

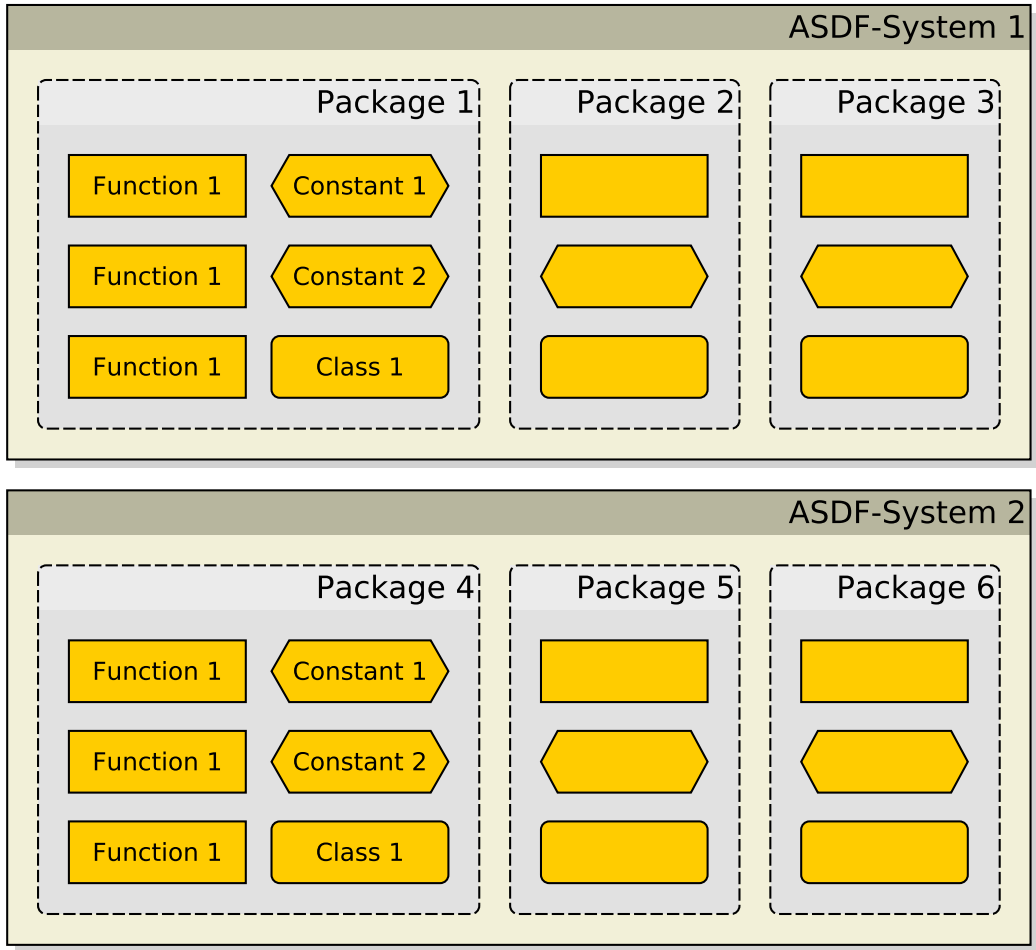
- ... manages different *systems* that contain packages
- ... marks which files are part of a package
- ... specifies dependencies
- ... informs about meta data (author, email, license, ...)

Extension Libraries

Short Interlude: ASDF

Important:

- Unique ASDF System names
- Unique package names
- Package namespaces → Functions, Constants, Classes not unique



Extension Libraries

Common Ground – Common Lisp

Functions we already know about come from Common Lisp

- Logic: and, or, not, ...
- Conditions: when, unless, if, case, cond, ...
- Singalling: handler-bind, unwind-protect, ...
- Math: +, -, /, *, sin, cos, ...
- Function handling: defun, defmethod, funcall, apply, ...

Not really an extension, more like the base. Central namespace: `CL-USER`

Extension Libraries

Alexandria

Based on the draft Alexandria documentation:
`http://common-lisp.net/project/alexandria/draft/alexandria.html`

Topics:

- Hash-tables
- `destructuring-case` (and `-bind`)
- Conditional '`let`' binding scopes: '`if-let`'
- More extended conditional code execution constructs
- Closure generation: '`curry`' and '`rcurry`'

Extension Libraries

Alexandria – Hash-Tables

The basics:

- Hash-tables are standard Common Lisp
- Alexandria extends functionality
- `http://cl-cookbook.sourceforge.net/hashes.html`

Extension Libraries

Alexandria – Hash-Tables

Defining and using hash-tables:

Example of using hash-tables

```
> (defparameter *my-hash* (make-hash-table))
> (setf (gethash 'my-entry-name *my-hash*) "Content")
> (format t "Hash contains: ~a~%"
          (gethash 'my-entry-name *my-hash*))

Content
```

Hash-tables override the `setf` setter and supply a specialized version.

Extension Libraries

Alexandria – Hash-Tables

Alexandria adds functionality like

- Copying hash tables: `copy-hash-table`
- Fallback to default value: `ensure-gethash`
- Extracting hash table information:
  - `hash-table-keys`: Return list of keys in table
  - `maphash-keys`: Apply function to every key
  - `maphash-values`: Apply function to every value
  - More...

Example of using maphash-keys

```
> (maphash-keys (lambda (x) (format t "Key: ~a~%" x)))
Key: key1
Key: key2
Key: key3
...
```

Extension Libraries

Alexandria – Destructuring-bind

Taking apart and parsing the inner structure of list constructs:

```
(destructuring-bind lambda-list expression &body body)
```

Is actually Common Lisp, but serves as a basis to `destructuring-case`.
Example:

Example of using destructuring-bind

```
> (destructuring-bind (a b (c d))
      (list 1 2 (list 3 4))
    (format t "~a␣~a␣~a␣~a~%" a b c d))

1 2 3 4
```

Extension Libraries

Alexandria – Destructuring-case

Case-based version:

```
(destructuring-case keyform &body clauses)
```

Combination of `destructuring-bind` and `case`:

Example of using destructuring-case

```
> (let ((x (list :foo 1 2)))
    (destructuring-case x
      ((:foo a b)
       (format t "First␣case:␣~a␣~a~%" a b))
      (((:alt1 :alt2) a)
       (format t "Second␣case~%"))
      ((:bar &key a b)
       (format t "Third␣case~%"))))

First case: 1 2
```

Extension Libraries

Alexandria – if-let

Signature:

```
(if-let bindings &body (then-form &optional else-form))
```

Only perform the body of a `let` when all bound values are bound as non-nil:

Example of using maphash-keys

```
> (if-let ((a 1)
           (b 2)
           (c (some-dynamic-function)))
    (format t "Success!~%"))
```

Output here is only generated when `c` is non-nil. `a`, `b` are non-nil by definition.

Extension Libraries

Alexandria – if-let's friends

Similar constructs exist:

- when-let
- when-let*
- switch

Extension Libraries

Alexandria – Flatten

Very useful:

`(flatten tree)`

With `tree` being a possibly nested list:

Example of using maphash-keys

```
> (flatten (list (list 'a 'b 'c) 2 (list 3 4 'test)))

(A B C 2 3 4 TEST)
```

Flattens a tree is lists to yield exactly one list level.

Extension Libraries
Alexandria

Interlude: **Closures – Detailed Definition**

"[...] a closure (also lexical closure or function closure) is a **function or reference to a function** together with a referencing environment—a table **storing a reference** to each of the **non-local variables** (also called free variables or upvalues) of that function. A closure—unlike a plain function pointer—allows a function to **access those non-local variables** even when **invoked outside its immediate lexical scope**."

`http://en.wikipedia.org/wiki/Closure_%28computer_programming%29`

Extension Libraries

Alexandria

Generating closures using ‘curry’:

- Closure = Function + Predefined Arguments + External Arguments

Example of defining a closure with curry

```
> (curry (lambda (x y)
           (format t "Defined␣in␣curry:␣~a~%" x)
           (format t "Funcall␣argument:␣~a~%" y))
         'my-curry-argument)
Result: #<CLOSURE (LAMBDA (&REST MORE) :IN CURRY) {100F674DDB}>
```

The closure carries:

- A lambda function
- A list of arguments

And is ultimately a **function**.

Extension Libraries

Alexandria

Generating closures using ‘rcurry’:

- Closure = Function + External Arguments + Predefined Arguments

Important difference: first, the external arguments, then the predefined

Example of defining a closure with rcurry

```
> (curry (lambda (x y)
           (format t "Funcall␣argument:␣~a~%" x)
           (format t "Defined␣in␣curry:␣~a~%" y))
         'my-curry-argument)
Result: #<CLOSURE (LAMBDA (&REST MORE) :IN CURRY) {100F674DDB}>
```

Extension Libraries

Alexandria

Using closures from the REPL: `funcall, apply`

Example of using curry closures with funcall

```
> (let ((clsr (curry (lambda (x y)
                       (format t "Defined␣in␣curry:␣~a~%" x)
                       (format t "Funcall␣argument:␣~a~%" y))
                     'my-curry-argument)))
    (funcall clsr 'my-funcall-argument))
Defined in curry: MY-CURRY-ARGUMENT
Funcall argument: MY-FUNCALL-ARGUMENT
```

Example of using curry closures with apply

```
> (let ((...))
    (apply clsr (list 'my-funcall-argument)))
Defined in curry: MY-CURRY-ARGUMENT
Funcall argument: MY-FUNCALL-ARGUMENT
```

Open Questions
Group Projects

Small group projects using the NXT+Lisp+ROS

- Groupsize: 3-4 People
- Common base to start from (will be on GitHub)
- Find team members, tell me about your team
- Must reflect what we talked about here
- Should solve a simple problem using the presented techniques
- Will be the basis for grades/passing

Open Questions
Group Projects

NXT Mindstorms Project Equipment:

- Actuators: 3x (Step-)Motors
- Sensors:
  - 1x Ultrasound
  - 1x RGB
  - 1x Sound
  - 2x Bumper
- Lots of cables
- Lots of bricks, Lego Technics parts, wheels, gears, ...