A WHIRLWIND INTRODUCTION TO REINFORCEMENT LEARNING WITH POEM/ALISP

MATTHIAS HÖLZL

Abstract. This tutorial contains a short introduction to the sublanguage of Poem that deals with probabilistic reasoning and reinforcement learning. We assume a certain level of mathematical maturity, but no prior knowledge of the specific techniques, tools or languages used.

1. Poem and Common Lisp

As its name implies, the Pseudo-Operational Ensemble Modeling Language (Poem), is a modeling and programming languag for ensembles. An important characteristic of ensembles is that they are designed to operate in a wide variety of open-ended, semi-structured or unstructured probabilistic environments; we call the set of all these environments the ensemble's *adaptation space*. Poem allows developers to design logical specification of the adaptation space and its behavior, to reason about the adaptation space using automated reasoners, to describe (possible incomplete) strategies for achieving solutions, and to employ hierarchical reinforcement learning techniques to automatically complete the strategies by finding good choices for the non-deterministic behaviors.

Poem is implemented as a domain-specific language based on Common Lisp, which integrates the Snark theorem prover, the ALisp system for hierarchical reinforcement learning, and extensions for manipulating logical theories, Markov decision processes, etc. Since this tutorial focuses on the reinforcement learning aspects of Poem, we ignore the parts of the language that deal with specifications in logic, reasoning and knowledge representation. Therefore, large parts of the code in the examples for this tutorial consists of Common Lisp expressions, so we start with a very short introduction to some features of the base language.

1.1. **Values.**

Lisp contains most of the usual kinds of values: numbers, strings, characters, etc. Some literals are written differently than in other languages. Fig. 1 shows some examples.

The boolean values are written `nil` and `t`; in conditionals, values of an arbitrary type can appear, and every value different from `nil` counts as a true value. Proper lists are either the empty list or a pair consisting of a first value (called the `car` of the list) and a second value which is itself a list (called the `cdr` and pronounced "could-er"). A peculiarity of Common Lisp is that the empty list `()` is identical to the false value `nil` (which is also a symbol). Furthermore, the empty list always evaluates to itself. Therefore `()` and `nil` can be used interchangeably, but it is good style to write `()` when the value is used as list and `nil` to denote the boolean value.

Common Lisp uses packages to partition the namespace. *Packages* can export symbols, thereby making them visible to other packages. There is always a notion of the "current package" which is the package that is used to intern symbols that do not include a package prefix.

Symbols are similar to "interned strings" in Java or "unique strings" in other programming languages, except that symbols "belong to" packages (i.e., each symbol is interned in a particular package). Like strings, symbols consist of a sequence of characters; unlike strings, symbols are immutable and two symbols in the same package are identical whenever their lexical representation is the same. To access an exported symbol in a package that is not directly visible from the current package it can be prefixed with the package name, i.e., to access the symbol `bar` in package `foo`, write `foo:bar`. If a symbol is not exported, it can still be accessed by separating the symbol and package names with two colons: `foo::bar` accesses symbol `bar` in package `foo` even when it is not exported. (This should only be used for debugging purposes).

Typically, symbols are written without any delimiters; they can contain a wide range of characters: letters, number, `-`, `+`, `*`, etc. However, if a symbol contains spaces (or other whitespace) it must be enclosed in vertical bars (`|`) otherwise it would be interpreted as several consecutive symbols. Unless a symbol is enclosed in vertical bars, the Lisp reader converts all letters in the symbol to upper case; therefore Lisp behaves like a case insensitive language (even though it is in reality case sensitive).

Keywords are symbols with some special features: they are interned in the `keyword` package and can be written in the form `:my-keyword`, i.e., starting with a colon. In contrast to other symbols, keywords are self evaluating, i.e., they don't have to be quoted, and they cannot be used as names of variables or functions. Keywords are often used as data objects since they don't have to be quoted and avoid some complications with packages; they can also serve as names for named function arguments (see below).

Uninterned symbols are yet another special kind of symbol (they don't call Lisp a symbolic programming language for nothing...). An uninterned symbol is, as the name implies, a symbol that is not interned in any package. Uninterned symbols can be written in the form `#:my-symbol`; each occurence of `#:my-symbol` gives rise to a fresh symbol that is not equal to anyting else, i.e.,

```
(eq 'my-symbol 'my-symbol)       ⇝ t
(eq :my-symbol :my-symbol)       ⇝ t
(eq 'my-symbol :my-symbol)       ⇝ nil
(eq #:my-symbol #:my-symbol)     ⇝ nil
```

Uninterned symbols are most commonly used to avoid lexical capture of introduced names in macros, and in package declarations since they don't pollute the namespace in which the package is defined.

Symbols and lists play an important role in Common Lisp: In Lisp, programs are themselves represented as Lisp data, with symbols and lists serving as representations for variables and function calls, respectively.

1.2. **Evaluation.** The source code of a Lisp program is stored in files, typically with the suffix `.lisp` or `.cl`. When the Lisp system processes a file it first converts the textual representation into a Lisp data structure; this data structure is then evaluated according to the following rules:[1]

- All objects except lists and symbols evaluate to themselves.
- Symbols represent (global or local) variables and are looked up in the variable environment.

[1]These rules are not really correct, but they should be precise enough to understand most programs.

Type	Example Literals
Boolean	`nil`, `t`
Number	`1`, `-123`, `1.23e2`
String	`"This is a string"`
Symbol	`print`, `a-1`, `*var*`, `|Symbol with space|`
Symbol in Package `a`	`a:my-symbol`, `a::internal`
Keyword	`:my-keyword`
Uninterned Symbol	`#:uninterned`
Character	`#\A`, `#\Newline`
List	`()`, `(1 2 3)`, `(print "Hello")`
Vector/Array	`#(1 2 3)`, `#((1 2) (3 4))`

FIGURE 1. Values

- Lists represent function calls, macros or applications of built-in operators (so-called *special forms*). Function calls are evaluated in the following manner:
  - Each element of the list is recursively evaluated. If the first element of the list is a symbol, its value is looked up in the *function environment*, all other symbols directly appearing in the list are looked up in the *variable environment*.
  - When all elements of the list are evaluated, the value of the first argument (a function) is applied to the other arguments.
  - The evaluation of a function call results in zero, one or more values.
- Each special form has its own evaluation rules. For example, `lambda`-forms evaluate to functions.

To prevent a symbol or list from being evaluated it can be prefixed by an apostrophe (`'`). For example `'x` evaluates to the symbol `x`, not to the value of the variable `x`. Similarly `'(print "x")` evaluates to a list with two elements (the symbol `print` and the string `"x"`), not to a function call. As a rule, lists that are used as data structures must always be preceeded by an apostrophe: `'(1 2 3)` evaluates to a list consisting of three integers, `(1 2 3)` leads to an error message, since Lisp tries to evaluate this form as a function call and `1` is not a valid function name.

1.3. **Global Variables and Functions.** Global variables are defined with the operators `defvar` and `defparameter`. By convention, the names of global variables start and end with an asterisk (`*`). For example:

```
(defvar *my-var* 123)
(defparameter *my-other-var* 234)
*my-var*                          ⇝ 123
*my-other-var*                    ⇝ 234
```

The value of global variables can be changed with the `setf` operator:

```
(setf *my-var 345)
(setf *my-other-var* 456)
*my-var*                          ⇝ 345
*my-other-var*                    ⇝ 456
```

The difference between `defvar` and `defparameter` is that a `defvar`-form does not overwrite an existing value for the variable whereas `defparameter` does. Thus, if we continue the example:

```
(defvar *my-var* 123)
(defparameter *my-other-var* 234)
```

```
*my-var*                          ⇝ 345
*my-other-var*                    ⇝ 234
```

Local variables can be bound with `let` and `let*` forms. The difference between these forms is that `let` binds all variables in parallel (so that none of the freshly introduced bindings is visible on the right hand sides) whereas `let*` binds the variables sequentially:

```
(let ((x 1)
      (y 2))
  (list x y))                                        ⇝ (1 2)

(let* ((x 1)
       (y (+ x 1)) ; not possible with let
  (list x y))                                        ⇝ (1 2)
```

This code also shows a comment; comments start after a semocolon (`;`) and end at the end of the line.

Functions are defined using the `defun` form:

```
(defun my-fun (x)
  (print x))
```

This expression defines a function called `my-fun` (i.e., it binds the function variable `my-fun` to the corresponding function object), that takes one argument. When this function is called it calls the `print` function to print the value of its argument:

```
(my-fun 123)
⇒ 123
⇝ 123
```

Each function returns the last value in its body; since the `print` function returns its argument after printing it, a call to `my-fun` also returns its argument. The `values` special operator can be used to return zero, one or more values:

```
(defun zero-values (x y)
  (print x)
  (print y)
  (values))

(defun three-values (x y)
  (print x)
  (print y)
  (values x y x))
```

Calling `zero-values` with arguments `1` and `2`, i.e., `(zero-values 1 2)`, prints `1` and `2` and returns no values, a call `(three-values 1 2)` prints `1` and `2` and returns the three values `1`, `2` and `1`. Multiple values can be bound with the form `multiple-value-bind`:

```
(multiple-value-bind (a b c) (three-values 1 2)
  (print a)
  (print b)
  (print c))
```

This code will print 5 lines of output: `1`, `2`, `1`, `2` and `1`. The first two lines are from the call to `three-values`, the last three lines from the `print` statements in the body of `multiple-value-bind`.

In addition to the required arguments, functions can take `optional`, `keyword` and `rest` arguments. Optional arguments need not be provided by the caller. The function definition can specify a default value for optional elements that are not provided, if no default value is specified, `nil` is used:

```
(defun opt-arg (x &optional (y 1) z)
  (format t "~&~A, ~A, ~A" x y z))
(opt-arg 'a)              ⇒ A, 1, NIL
(opt-arg 'a 'b)           ⇒ A, B, NIL
(opt-arg 'a 'b 'c)        ⇒ A, B, C
```

The function `format` prints formatted output to a stream. Here the stream is specified as `t`, denoting the standard output. The format directive `~&` is a (conditional) newline, `~A` takes the next unprocessed argument and prints it. In the first call to `opt-arg`, the call provides no values for the variables `y` and `z`, so their default values are used. In the second call a value is provided for `y` but not ,for `z`, in the last call, all variables are provided values by the caller.

If a function takes many arguments, calls to the function can be rather confusing. Keyword arguments can be used to clarify the roles of the arguments: like optional arguments keyword arguments need not be provided by the caller, but in calls, keyword arguments are explicitly named by a keyword and can therefore be provided in any order:

```
(defun key-arg (x &key (y 1) z)
  (format t "~&~A, ~A, ~A" x y z))
(key-arg 'a)                   ⇒ A, 1, NIL
(key-arg 'a :y 'b)             ⇒ A, B, NIL
(key-arg 'a :y 'b :z 'c)       ⇒ A, B, C
(key-arg 'a :z 'c :y 'b)       ⇒ A, B, C
```

1.4. Structures and Classes.

Common Lisp contains CLOS (Common Lisp Object System), an extremely powerful object system. There are two different class-like data types: Structures (also called structure-classes) and classes. Structures as well as classes contain only data members; methods are defined outside of the user-defined data types.

Structures have several restrictions that make them much less flexible than classes, but also more efficient. For example, classes can be redefined (and existing instances using the old class definition can be upgraded to the new class definition), classes support multiple inheritance, and the class of a class-instance can be changed dynamically. Structures provide none of these features, but they can therefore be implemented more efficiently than classes. Therefore it is advisable to use structure types only in situation where performance considerations are paramount.

Unfortunately, for historic reasons, the syntactic for of class and structure definitions is quite different. Structures are defined in the following way:

```
(defstruct (simple-state (:conc-name #:simple-))
  (start-loc '(0 0) :type list)
  robot-loc
  env)
```

This defines a structure-class `simple-state` with three instance variables that can be accessed using functions called `simple-start-loc`, `simple-robot-loc` and `simple-env`. The prefix of the instance accessors is defined by the `:conc-name` struct-option. Instances are created using the constructor `make-simple-state` which takes a keyword argument for each instance variable. The instance variable `start-loc` is additionally provided with a default value and restricted to values of type `list`. The call

```
(make-simple-state :start-loc '(1 2) :end-loc '(5 7) :env *my-env*)
```

creates a new instance of type `simple-state` in which the instance variables (which are typically called *slots* in Lisp) are initialized to the given values.

A class definition has the following form:

```
(defclass <simple-env> (<fully-observable-env> <grid-world>)
  ((move-success-prob :type float
                      :initarg :move-success-prob :initform 0.95
                      :accessor move-success-prob)
   (wall-collision-cost :initarg :wall-collision-cost :initform 0.5
                        :accessor wall-collision-cost)))
```

This defines a class `<simple-env>` (the use of angle brackets around the class name is a convention that is used in ALisp and has no further significance). This class inherits from two superclasses, `<fully-observable-env>` and `<grid-world>` and has two slots `move-success-prob` and `wall-collision-cost`. The name of the accessor functions and the keyword arguments for the constructor have to be explicitly specified using the `:accessor` and `:initarg` keyword arguments in the slot specification; furthermore each slot specifies a default value (`:initform`); the `move-success-prob` slot additionally specifies the type of values it can store. The accessors of classes defined with `defclass` are used as specified, no prefix is appended.

Instances of classes are created using the `make-instance` function, which takes the keywords specified in the class definition and the keywords inherited from superclasses. For example:

```
(make-instance '<simple-env> :move-success-prob 0.8)
```

(Note that we pass the *name* of the class as first argument to `make-instance`, i.e., the first argument to `make-instance` is typically quoted.)

1.5. **Generic Functions and Methods.** Methods do not belong to classes, instead they are grouped into *generic functions.* A generic function can explicitly be defined with `defgeneric` or implicitly by a method definition.

The code

```
(defgeneric description (thing))
```

defines a generic function `description` that takes a single argument. Methods can be specialized on this argument:

```
(defmethod description (thing)
  (print "Some unspecified thing"))
(defmethod description ((l list))
  (print "A list"))
(defmethod description ((n number))
  (print "A number"))
(defmethod description ((st simple-state))
  (print "Our very own state"))
(defmethod description ((env <simple-env>))
  (print "A simple environment"))

(description "Foo")                         ⇒ Some unspecified thing
(description '())                           ⇒ A list
(description 123)                           ⇒ A number
(description (make-simple-state))           ⇒ Our very own state
(description (make-instance <simple-env>)) ⇒ A simple environment
```

Command	Meaning
`C-x C-f`	Find file (also used to create a new file)
`C-x C-s`	Save current buffer
`C-x o`	Switch to other buffer
`C-x 1`	Hide other buffers
`C-x 0`	Hide this buffer
`C-c C-k`	Compile buffer
`C-c C-c`	Compile current definition
`C-c C-z`	Switch to Lisp interaction buffer
`C-M-i`	Complete current input
`M-.`	Go to definition
`C-c i`	Inspect element (non-standard)

Figure 2. Some Emacs commands

Note that methods can be defined on "primitive" types (such as number), on structures and on classes. It is even possible to define methods specialized on single objects:

```
(defmethod description ((n (eql 0)))
  (print "Naught"))
(description 0)                                  ⇒ Naught
```

Generic functions support multi-dispatch, i.e., they can be specified on several arguments:

```
(defmethod multi (x y)
  (list x y))
(defmethod multi ((x number) (y number))
  (+ x y))
(defmethod multi ((x string) (y string))
  (concatenate 'string x y))
(multi  'a  'b)                                 ⇝ (A B)
(multi 'a   1)                                  ⇝ (A 1)
(multi  1   2)                                  ⇝ 3
(multi 'a  "b")                                 ⇝ (A "b")
(multi "a" 'b)                                  ⇝ ("a" B)
(multi "a" "b")                                 ⇝ "ab"
```

2. Intermezzo: Some Useful Emacs Commands

Emacs with the Slime mode for Lisp development offers many features to navigate through source code and to interact with a running Lisp process. You start the Lisp listener with the Emacs command `M-x slime`.

Some of the most useful are given in Fig. 2. Some commands that are only valid in the interaction buffer are given in Fig. 3.

3. ALisp Extensions

ALisp adds several new operators to Common Lisp that allow the learning mechanism to work. Programs that contain any of these operators are called *partial programs*.

The most important ones are the following:

action: Performs an action on the environment. A label has to be provided before the action name.

Command	Meaning
`M-p`	Previous command
`M-n`	Next command
`,in`	Change package
`,compile-system`	Compile an ASDF system
`,load-system`	Load an ASDF system
`,force-compile-system`	Force compilation of a system
`,force-load-system`	Force (re)loading of a system
`,sayoonara`	Quit the Lisp process

FIGURE 3. Some Emacs commands in the interaction buffer

call: Calls a partial program with arguments. Can optionally have a label that is used by the learning algorithms to determine which call sites should be learned together; if no label is provided, the name of the function is used. For example,

```
(call (nav some-location))
```

calls the partial program `nav` with argument `some-location` (which is the location we want to travel to, and should be bound as variable).

with-choice: Selects a value from a set of values. A label is mandatory to allow the learning algorithms to determine how the choice should be learned. E.g., choose among the action values `N`, `E`, `S`, `W` and then perform the corresponding action:

```
(with-choice navigate-choice (dir '(N E S W))
  (action navigate-move dir))
```

choose: Chose (non-deterministically, i.e., learn to choose) between several different actions or partial programs. As usual, a label has to be provided. For example

```
(choose choose-waste-removal-action
        (call (pickup-waste))
        (call (drop-waste)))
```

(The difference between `with-choice` and `choose` is that the first one chooses *values* whereas the second one chooses *behaviors* to perform. Of course, since behaviors are first-class values, `with-choice` and `choose` can be used somewhat interchangeably. For example, the previous action choice could also be represented using the `choose` operator with four actions in the body.)

4. The Simple-Waste Example

The Simple-Waste example is meant to introduce ALisp using the simple scenario of a robot moving toward a target in an arena in which there may be some obstacles. The following sections assume that your Lisp environment is already correctly set up.

4.1. Running the Simple-Waste Example.

In the Lisp listener (REPL), evaluate the following forms:

```
CL-USER> (asdf:load-system :waste)
[...]
CL-USER> (in-package :simple-prog)
#<PACKAGE "SIMPLE-PROG">
SIMPLE-PROG> (explore-environment)
Welcome to the simple robot example.
```

```
This environment demonstrates a robot that moves around on a
rectangular grid, until it reaches a target area.  X's on the map
represent walls, blank spaces are roads.  The robot is represented
by 'r'.  You can move by entering N, E, S, W.  To quit the
environment, enter NIL.  (All input can be in lower or upper case.)

Last observation was
XX0X1X2X3XXX
0X        0X
1X  rr    1X
2X        2X
XX0X1X2X3XXX
Target: (0 0)
Action?
```

The `load-system` form loads the `:waste` system definition which contains, among others, the code for the Simple-Waste example. The `explore-environment` runs a function that allows you to interactively explore the environment. The start position of the robot is randomly generated and may be different for each execution. After completing an episode or entering `nil` at the prompt, you can start the reinforcement learner by calling `learn-behavior`:

```
SIMPLE-PROG> (learn-behavior)
Learning behavior using random exploration strategy
Learning
Episode 0.
NIL
```

The `learn-behavior` function calls the primitive `learn` function with parameters that are controlled by its keyword arguments and some global variables. After learning is completed, you can evaluate the performance of the learned policy against environments with randomly genereated robot start positions:

```
SIMPLE-PROG> (evaluate-performance)

Learning curves for HORDQ-A-1, HORDQ-A-2 are:
Evaluating policies...........................................
Evaluating policies...........................................
#((#(-2.36 4.71 4.71 4.72 4.73 4.7 4.7
     4.73 4.72 4.72 4.71 4.73 4.73 4.71
     4.71 4.72 4.69 4.72 4.71 4.69 4.7
     4.71 4.72 4.71 4.68 4.71 4.7 4.7
     4.72 4.72 4.71 4.73 4.72 4.72 4.73
     4.69 4.71 -9.66 -8.93 -8.96 4.72
     4.71 4.72 4.71 4.74 4.7 4.72 4.71 4.71 4.71))
  (#(-3.52 1.31 3.66 4.53 2.06 0.86 1.27
     2.62 4.74 4.72 4.72 4.71 4.69
     4.72 4.7 4.69 4.65 4.72 4.67 4.68
     4.67 4.68 4.68 4.69 4.72 4.7 4.71
     4.7 4.72 4.73 4.73 4.72 4.72 4.69
     4.71 4.72 4.71 4.71 4.69 4.72 4.73
     4.72 4.72 4.71 4.69 4.69 4.72 4.72 4.71 4.71)))
; No value
```

The function `learn-behavior` stores copies of the policies learned after having completed 2%, 4%, . . . , 100% of the steps in the training run. `evaluate-performance` runs each of these policies against randomly generated examples and returns a vector of the resulting scores. In the example, the reward for reaching the target field is 5.0 and the cost for each step is 0.1. The cost for bumping into a wall is 0.5. To complicate the problem, the robot only moves into the desired direction with 90% probability, and perpendicular to this direction otherwise. Hence, the best expected score in the environment is ≈ 4.7, which both algorithms achieve frequently. Not that the first algorithm rapidly achieves this score for most of the runs, but several runs toward the end of the learning curve show severely degraded performance. The second algorithm converges less rapidly, but consistently stays near the maximum performance after about a quarter of all tries. We will see why the algorithms exhibit this behavior when we look at their implementation.

Let's try a slightly more involved example:

```
SIMPLE-PROG> (learn-behavior :environment-type :medium
                             :use-complex-environment t)
Learning behavior using random exploration strategy
Learning
Episode 0.......................................
NIL

SIMPLE-PROG> (evaluate-performance)
Learning curves for HORDQ-A-1, HORDQ-A-2 are:
Evaluating policies..................................................
Evaluating policies..................................................
#((#(-6.85 -21.16 -3.87 -3.01 -3.48 -4.41
     -3.73 -21.15 -4.03 -21.56 -4.09
     -4.51 -3.82 -23.64 -24.11 -4.12 -3.19
     -4.31 -3.65 -3.09 -3.83 -2.48
     -2.56 -3.94 -2.68 -3.86 -2.29 -21.24
     -4.26 -4.21 -4.51 -4.51 -3.49
     -21.61 -3.23 -2.07 -2.98 -2.04 -2.33
     -2.46 -2.03 -2.93 -2.41 -4.12
     -4.35 -3.94 -4.61 -2.89 -4.6 -4.12))
  (#(-6.56 -6.23 -4.18 -4.44 -4.35 -4.28
     -4.07 -4.02 -4.12 -4.32 -4.02
     -3.73 -4.04 -0.16 -0.54 0.04 -0.47
     0.42 -0.37 1.08 1.1 1.39 0.26 0.7
     0.35 -0.66 0.49 1.07 2.91 3.95 2.96
     2.61 2.5 3.67 3.1 3.05 3.36 3.57
     3.49 3.99 2.84 3.82 3.69 3.9 3.58
     3.97 3.98 4.08 3.97 4.02)))
; No value

SIMPLE-PROG> (explore-environment)
Welcome to the simple robot example.

This environment demonstrates a robot that moves around on a
rectangular grid, until it reaches a target area.  X's on the map
represent walls, blank spaces are roads.  The robot is represented
by 'r'.  You can move by entering N, E, S, W.  To quit the
environment, enter NIL.  (All input can be in lower or upper case.)
```

```
Last observation was
XX0X1X2X3X4X5X6X7XXX
0X      XX        0X
1X      XX        1X
2XXXXX  XX        2X
3X      XX        3X
4X        rr      4X
5X                5X
6X                6X
7X                7X
XX0X1X2X3X4X5X6X7XXX
Target: (0 0)
Action? nil
```

Here we explore a medium-sized environment with a slightly more complex structure, with the same reward structure as before. To compensate for this increased complexity, `learn-behavior` runs the experiment with a larger number of steps, as can be seen from its output (a dot is printed for every 2500 steps). The evaluation shows that in this case the first algorithm does not learn a useful behavior, whereas the second algorithm again converges to a nearly optimal policy (taking into account the increased size of the arena and the additional movement steps necessary to drive around the wall when starting in the upper right quadrant). To investigate the behaviors of policies it is sometimes useful to observe them in action. This can be done by calling `(explore-policies)`:

```
SIMPLE-PROG> (explore-policies)
Welcome to the simple robot example.
[...]
Env state:
XX0X1X2X3X4X5X6X7XXX
0X      XX        0X
1X      XX        1X
2XXXXX  XXrr      2X
3X      XX        3X
4X                4X
5X                5X
6X                6X
7X                7X
XX0X1X2X3X4X5X6X7XXX
Target: (0 0)
Stack: ((NAV NAVIGATE-CHOICE ((LOC 0 0))) (TOP NAV NIL))>
Set of available choices is (N E S W)
--------------------------------------------------
Advisor 0:
Componentwise Q-values are
  #((N (Q -74.13) (QR -0.2) (QC -73.93) (QE 0.0))
    (E (Q -74.1) (QR -0.17) (QC -73.93) (QE 0.0))
    (S (Q -74.03) (QR -0.11) (QC -73.93)(QE 0.0))
    (W (Q -74.03) (QR -0.1) (QC -73.93) (QE 0.0)))
Recommended choice is W
--------------------------------------------------
Advisor 1:
Componentwise Q-values are
  #((N (Q -0.79) (QR -0.1) (QC -0.69) (QE 0.0))
    (E (Q -0.79) (QR -0.1) (QC -0.69) (QE 0.0))
    (S (Q -0.61) (QR -0.1) (QC -0.51) (QE 0.0))
    (W (Q -0.71) (QR -0.1) (QC -0.61) (QE 0.0)))
Recommended choice is S
--------------------------------------------------
Please enter choice, or nil to terminate.
```

In this interaction we have elided the first step (which is a choice point that provides the only alternative `no-choice`. Simply type `no-choice` at the prompt. Like `explore-environment`, the function `explore-policies` allows us to interact with the environments. In addition it shows the Q-functions computed by the different learning algorithms and their choice. After following the advice of the first algorithm for a few steps we see why it performs badly on many examples: even though the robot bumps into the wall when moving west, the algorithm repeatedly suggests this move. The second algorithm, in contrast, correctly suggests moving south. (When looking at the code we will see that the first algorithm operates without knowing the robot's position on the board or the walls adjacent to the robot's position so that it cannot develop a strategy that behaves sensibly if there are obstacles in the arena.) As a final example, let us try the second algorithm on a maze-like example:

```
SIMPLE-PROG> (learn-behavior :environment-type :maze
                             :algorithm-names '(hordq-a-2))
Learning behavior using random exploration strategy
Learning
Episode
0...........................................................
 ...........................................................
 ...........................................................
 ...........................................................
 ...........................................................
 ...........................................................
 ..............................................
NIL

SIMPLE-PROG> (evaluate-performance)
Learning curves for HORDQ-A-2 are:
Evaluating policies.................................................
#((#(-7.63 -7.08 -4.17 -4.15 -2.31 -1.45
     -1.17 -0.27 0.75 0.82 1.61 2.89
     2.26 2.79 2.45 3.01 2.78 2.78 2.96
     2.94 2.78 2.8 2.9 2.89 2.77 2.66
     3.01 2.92 2.68 2.96 2.82 2.83 2.87
     2.87 2.92 2.64 2.77 2.87 2.59 2.75
     2.95 2.82 3.13 2.7 3.07 2.65 2.87 3.02 2.73 2.77)))
; No value

SIMPLE-PROG> (explore-policies)
[...]
XX0X1X2X3X4X5X6X7X8X9XXX
0X      XX            0X
1X      XXXXXXXXXX  XX1X
2XXXXX  XX  XX        2X
3X      XX  XXXXXXXX  3X
```

```
4X  XXXXXX  XX        4X
5X          XX  XXXXXX5X
6XXXXXXXXX  XX    rr  6X
7X  XX      XXXX      7X
8X  XX  XX  XX  XX    8X
9X      XX            9X
XX0X1X2X3X4X5X6X7X8X9XXX
Target: (0 0)
```

Since the average path length in this environment is ≈ 14 steps, the policy is again close to the optimal one.

4.2. **Diving into the Code.** Let us now look at the code of the Simple-Waste example. The system definition is in the file `Sources/waste.asd`. The system definition contains some administrative information and then a list of the source files that give rise to the program. The ones comprising the Simple-Waste example are the ones whose name starts with `simple-`; let us look at them in turn.

4.2.1. *The File `simple-package.lisp`.* This file contains few surprises: it defines two packages, `simple-env` for the definition of the environments, and `simple-prog` for the definition of the program and the example driver. The code is split into two packages, since ALisp defines some names with incompatible definition in several packages so that we cannot import all required packages from ALisp into a single package.

4.2.2. *The File `simple-env.lisp`.* The file `simple-env.lisp` contains the definition of the environmen, i.e., the Markov Decision Process on which the partial program operates. Although ALisp provides a more declarative way to define environments as a form of dynamic Bayesian networks (2TBNs), I have opted to implement the environment of the Simple-Waste example manually, so that its behavior is more immediately visible.

In ALisp the environment is a class derived from `<env>`, and if it is fully observable from the subclass `<fully-observable-env>`. The state of an environemnt is represented by a class whose instances are assumed to be immutable. Therefore we define a structure `simple-state` containing the state variables, and a class `<simple-env>` that inherits from `<fully-observable-env>` and provides the persistent information about the environment. Since we operate on a grid world, `<simple-env>` also inherits from `<grid-world>` so that it can make of the useful functionality predefined in that class, e.g., the computation of shortest paths according to Floyd's algorithm.

The structure *simple-state* contains three slots: the start location of the robot, the current location of the robot and a back-pointer to its environment. It defines methods for cloning states (`clone`), for equality testing (`same`) and a canonical representation (`canonicalize`). Furthermore its `print-object` method prints an ASCII-Representation of the grid-world when the variable `*print-graphically*` has a true value.

The class `<simple-env>` contains instance variables for the probability that a move succeeds (`move-success-prob`), for the cost of hitting a wall (`wall-collision-cost`), for the cost of each step irrespectively of the action (`cost-of-living`) for the target location (`target-loc`) and for the reward obtained when reaching the target location (`final-reward`). Additionally it contains a function that provides a randomly chosen, valid start location (`start-loc-sampler`) which, by default, is set to a function that uniformly choses from all valid locations.

The function `reward` computes the reward obtained by performing a single step. It uses the auxiliary functions `is-terminal-state` so that a zero reward is given for all steps taken after a process has entered a final state, and `move-would-hit-wall-p` to check whether a desired move would hit a wall. Similarly, the function `compute-next-loc` computes the next location, given an environment, a state and an action, taking into account the possibility that the robot may slip perpendicular to its intended direction.

The method `sample-next` combines the results of `compute-next-loc` and `reward`; this generic function is used to "drive" the MDP. The function `sample-init` returns a randomly chosen initial state. The functions `make-simple-env-`n create environments that can be used for experiments.

4.2.3. *The File* `simple-prog.lisp`*.* The program is very simple. It defines accessor functions `robot-loc` and `robot-env` for the location and environment of the robot, and a top-level function `simple-robot-prog` that simply calls the navigation strategy `nav`. The partial program `nav` repetedly moves in one of the four allowed directions until it has reached the desired location.

4.2.4. *The File* `simple-features.lisp`*.* This file defines several features that can be used for function approximations, and several "3-part-Q" featurizers that can be used to perform HORD-Q learning. The first featurizer, `*simple-featurizer-0*` extracts only the current choice, so that function approximations using this featurizer cannot base decisions on the location of the robot or on the walls adjacent to the robot's location. This featurizer almost always leads to very bad learning performance.

`*simple-featurizer-1*` has slightly more information; in addition to the chosen action it can also access the valid target directions. This featurizer generally performs well in arena without obstacles but degrades significantly as soon as only a few obstacles are in the arena.

The third and fourth featurizer take the percise location of the robot into account; they work well in most cases, although they need significant training data.

4.2.5. *The File* `simple-example.lisp`*.* This file contains definitions of variables and constants that (hopefully) simplify the interaction with the reinforcement learning system.

4.3. **Tasks.** The Simple-Waste example can be enhanced in many ways. Some useful tasks to try are the following:

(1) Experiment with the predefined featurizers in the different environments. Why do some featurizers work well in open environments and not in ones with obstacles?
(2) Explore additional featurizers. For example, the `grid-world` package provides a `shortest-path` function that can be used to (efficiently, i.e., in $O(n^3)$) compute shortest paths using Floyd's algorithm. Can this information be used to build a better featurizer?
(3) Allow the target location to vary stochastically. Extend the environment model (and possibly the program and featurizers) to cope with this situation. Can the extensions from (2) be useful in this case?
(4) Have several different target locations, possibly with different rewards. See whether the `shortest-path-dist` function from the `grid-world` package can be useful for featurizers in this case.
(5) Introduce fields with different costs: Moving along the main hallway should be less expensive than moving inside stalls.
(6) Introduce slanted or windy fields that cause the robot to deviate from its path with a high probability. What effect do these fields have on the

performance of (2), what additional "sensors" could be used by a featurizer to cope with this behavior.

(7) Provide the robot with a limited amount of fuel that is used up by performing the various actions; give a large negative reward for running out of fuel.

(8) Introduce waste items into the environment and base the reward for the robot on the number of waste item it has picked up and carried to a target area. Initially introduce a single waste item at a random position when the environment is created. (A hierarchical program structure with multiple choice nodes is very useful for this case.)

(9) Extend the environment of (8) so that new waste is dropped in the environment while the robot is operating.

(10) Extend the environment from (9) so that the reward is partially based on the level of cleanliness of the environment. Introduce an action that allows the robot to sleep for a predetermined time without incurring any cost of living. What information would a featurizer have to provide so that the robot can learn a good strategy for sleeping?

(11) Extend the example to multiple robots.