Gridworld Framework Manual

Daphne Liu

January 18, 2011

1 Introduction

The Gridworld Framework is a LISP-based implementation of a self-motivated cognitive agent framework devised by Len Schubert and Daphne Liu. The self-motivated cognitive agent implemented in the framework thinks ahead, plans and reasons deliberately, and acts reflectively, both by drawing on knowledge about itself and its environment and by seizing opportunities to optimize its cumulative utility.

2 Getting Started

To experiment with the Gridworld Framework, you must have the source files **gridworld-planning.lisp**, **gridworld-definitions.lisp**, and **simulation-and-function-evaluation.lisp**. With the LISP routines already defined in these files, you can program the knowledge, planning and behavior (physical and dialog) of a simulated motivated, mobile, exploratory agent in a graphical roadmap. For notational generality, we call such an agent *AG* in the code (we named it *ME* for *motivated explorer* in publications listed in Section 5). You are advised to read the header comments in these files, as well as detailed comments of the LISP routines and global variables you will use in these files.

2.1 Syntax

If you are new to LISP, please review a comprehensive guide to LISP syntax, such as the one available at the Webpage `http://www.csci.csusb.edu/dick/samples/lisp.syntax.html`

LISP accepts almost all alphanumerical strings for predicate names and function names; while a predicate name or function name cannot be a single dot or a number, syntactically speaking, it can be any string of non-space and non-delimiter characters. That said, you should abide by the following rules when choosing a name for a *predicate*:

1. It should be easily translatable into English (see details in Section 2.3.1);
2. Any verb(s) appearing in it should be in the third person, singular, present tense.

A *literal* (lit) is either an atom (predication) enclosed in parentheses of the form (*pred* t_1 t_2 ... t_n) where *pred* is an n-ary predicate name and $t_1, t_2, ..., t_n$ are *terms*, or the negation of such an atom (*not* (*pred* t_1 t_2 ... t_n)).

A *term* is either a number, constant, variable, function, or a *reified predication*. We adopt LISP's inherent definitions of numbers, constants, variables, and functions here, except that you should prefix each variable name with a ? symbol to indicate its being a variable in the framework. A *reified literal* has either the form (*whether lit*) or the form (*that lit*) with *lit* being a literal; it typically appears as the object argument of a *wants* or *knows* predicate. For instance, (*knows AG* (*whether* (*is_bored AG*))) expresses that *AG* knows whether *AG* is bored, and (*wants AG* (*that* (*eats AG pizza*))) states that *AG* wants to eat *pizza*.

We further distinguish some terms as being *evaluable terms*. These are functions or predicates whose operators are among $+$, $-$, $*$, $/$, $<$, $<=$, $=$, $>=$, $>$, *random*, and user-defined (i.e., defined by you) function or predicate names ending in ?. Evaluable terms often appear in operator specifications and are used in the procedural attachment technique detailed in Section 2.4.1.

For each *action operator* you design in the Gridworld Framework, you need to define both its *model version* and its *actual version*. We elaborate on both the semantics and the reasons for this distinction in Section 2.4. The following is the schema for the *model version* of an *action operator* named *action_name*, with requisite fields : *name*, : *pars*, : *preconds*, : $effects$, : $time-required$, and : *value*:

```
(setq action_name
   (make-op
        :name 'action_name
        :pars '(?var1 ?var2 ... ?varn)   ; where ?vari's are variables
        :preconds '( litp1 litp2 ... litpn ) ; where litpi's are lits
        :effects '( lite1 lite2 ... liten )  ; where litei's are lits
        :time-required 'term1  ; where term1 evaluates to some number
        :value 'term2          ; where term2 evaluates to some number
   )
)
```

The following is the schema for the corresponding *actual version* of the action operator, with requisite fields : *name*, : *pars*, : *startconds*, : *stopconds*, : *deletes*, and : *adds*:

```
(setq action_name.actual
   (make-op.actual
        :name 'action_name.actual
        :pars '(?var1 ?var2 ... ?varn)   ; where ?vari's are variables
        :startconds '( ltstt1 ltstt2 ... ltsttn ) ; where ltstti's are lits
        :stopconds '( ltstp1 ltstp2 ... ltstpn ) ; where ltstpi's are lits
        :deletes '( ltd1 ltd2 ... ltdn ) ; where ltdi's are lits
        :adds '( lta1 lta2 ... ltan ) ; where ltai's are lits
   )
```

)

To experiment with question-answering in the Gridworld Framework, you will need to familiarize yourself with the syntax for posing questions (as the user) to *AG*. Let *lit* denote a literal of either the form (arg_1 *pred* arg_2 ... arg_n) or the form (NOT (arg_1 *pred* arg_2 ... arg_n)), where arg_1 is the subject argument of the n-nary predicate relation *pred*. A wh-question to *AG* has the form (*ask−wh lit*), and a yes/no-question to *AG* has the form (*ask−yn lit*). To input questions to *AG*, type and enter command (*listen*!) at the prompt. After issuing command (*listen*!), if the user has no input to impart, then the user should simply enter () at the prompt and then hit the enter key. Otherwise, an example user input would be ((*ask−yn* (*AG is_bored*)) (*ask−wh* (*AG likes* ?*wh*))). Details and extensive examples are in Section 3.2.

2.2 Defining and Creating a Roadmap

You can define a set of named locations, as well as named roads connecting some or all of these locations. Roads start at, pass through, and end at specified locations, and named roads and named locations have atomic labels. In addition, you can define and place animate and inanimate entities of specified types at various locations. For animate entities (agents), the specified ground literals may include ones like beliefs and preferences. The types assigned to entities are separately specified, allowing shared properties of entities of that type to be listed. Whereas *AG* is mobile and exploratory, interacting with the user and other entities while striving to maximize its cumulative utility, all other entities are stationary and only reactive in their interactions with *AG*.

To define the roadmap of your Gridworld, use the **def-roadmap(points, roads)** routine defined in **gridworld-definitions.lisp**. For example, the following defines and creates a roadmap with named locations *home*, *grove* and *plaza*, and named roads *path*1 and *path*2. Furthermore, *path*1 connects *home* and *grove*, which are 3 distance-units apart; while *path*2 connects *home* and *plaza*, which are 2 distance-units apart.

```
(def-roadmap '(home grove plaza)
             '((path1 home 3 grove) (path2 home 2 plaza)))
```

This *roadmap* definition would add the following specific facts to *AG*'s knowledge base:

- (*point home*), (*point grove*), (*point plaza*)
- (*road path*1), (*road path*2)
- (*navigable path*1), (*navigable path*2)
- (*is_on home path*1), (*is_on grove path*1), (*is_on home path*2), (*is_on plaza path*2)

The above definition, however, does not automatically add connectivity predications to *AG*'s knowledge base. Rather, connectivity information, including length and road name, between each point and its immediate neighbor is directly stored in the *next* property of each point. Therefore, the *next* properties of *home*, *grove*, and *plaza* are ((*path*1 3 *grove*) (*path*2 2 *plaza*)), ((*path*1 3 *home*)), and ((*path*2 2 *home*)), respectively.

2.3 Defining and Creating Types and Entities

Specific entities are defined and placed in your Gridworld, in two stages. First, you need to define some types of entities, and their associated general, permanent properties. This is done using the LISP routine **def-object(obj-type, properties)** of file **gridworld-definitions.lisp**. Then you apply a function to create a named entity of a particular type, placing it in your Gridworld along with its associated things, and some facts about its current state and about its current propositional attitudes. The function call is of form **place-object(name, obj-type, point, associated-things, curr-facts, propos-attitudes)** in file **gridworld-definitions.lisp**.

2.3.1 Defining an Entity Type

In defining various types of animate and inanimate entities using **def-object(obj-type, properties)**, you associate some general, permanent properties with those entities. Furthermore, in making those definitions, this general knowledge is automatically added to *AG*'s knowledge base.

For example, consider the following type definition which specifies the (abbreviated) general properties *is_animate*, *is_furry*, (*has_IQ* 50) for the type *sasquatch*:

```
(def-object 'sasquatch '(is_animate is_furry (has_IQ 50))
```

This sasquatch type definition would add the following (expanded) general facts to *AG*'s knowledge base:

```
((sasquatch ?x) => (is_animate ?x))
((sasquatch ?x) => (is_furry ?x))
((sasquatch ?x) => (has_IQ ?x 50))
```

You should choose and create predicate names that can be easily rendered into English. The rendering of predications into English has been implemented in the Gridworld Framework via the **verbalize(wff)** and related routines in **simulation-and-function-evaluation.lisp**. A positive literal is translated into an English sentence by pulling out the subject (i.e., the first argument of the predicate) and placing it first, by replacing underscores and dashes and question marks with spaces, and by replacing plus signs with arguments in the list of arguments after the first argument of the predicate. For any predicate of arity 2 or higher, its last argument is always inserted at the end of the rendered English sentence, so you need not append a question mark to the end of its predicate name unless you intend for it to appear as an evaluable term somewhere in your code. A negative literal is translated in the same way except that the *not* operator is verbalized by prepending "it is not the case that" to the verbalization of the remainder of the literal.

For instance, (*not* (*has_IQ Grunt* 50)) is verbalized as "it is not the case that Grunt has IQ 50." The literal (*is_at AG* (*the_point+units_from+towards+on_road*? ?*d* ?*x* ?*y* ?*z*))) is rendered as "*AG* is at the point *d* units from *x* towards *y* on road *z*."

2.3.2 Defining and Placing an Entity

In naming an entity of a specified type and placing it at some point in the Gridworld, using **place-object(name, obj-type, point,associated-things, curr-facts, propos-attitudes)**, you supply three kinds of additional information for it:

1. entities that it currently has; These may be regarded as possessions in the case of animate beings, or as contained or attached objects in the case of inanimate objects such as trees or boxes; e.g., a type predication like (*key* $Key3$) supplied under this heading means that the named entity has $Key3$ of type *key*;

2. current state facts about it; e.g., (*is_hungry Grunt*), or (*likes Grunt Tweety*);

3. propositional attitudes such as (*knows Grunt* (*that* (*has AG* $Banana1$))), or (*wants Grunt* (*that* (*has Grunt* $Banana1$))); It is even possible to have nested knowledge facts or goal facts such as (*knows Grunt* (*that* (*knows AG* (*that* (*wants Grunt* (*that* (*has Grunt* $Banana1$))))))), i.e., *Grunt* knows that *AG* knows that *Grunt* wants to have $Banana1$.

For an animate entity, we may assume that it knows all of its current facts – they are open to its introspection. Therefore, we are assuming that animate entities know what kinds of things they are, what they have, what their current state is, what they know, and what they want. Note that even inanimate entities may have propositional attitudes associated with them. For example, for a certain box, Box1, we might have (*contains_message* $Box1$ (*that* (*is_located_at* $Key3$ *Grove*))). The difference is just that the inanimate entity doesn't know this fact.

2.4 Defining and Creating an Action Operator

Operators, or action types, are defined for *AG* in a STRIPS-like syntax, with a list of parameters, a set of preconditions and a set of effects, an anticipated net utility value, and an expected duration. Consider the following operator *sleep* with formal fatigue and hunger level parameters $?f$ and $?h$, respectively:

```
(setq sleep
   (make-op
   :name 'sleep
   :pars '(?f ?h)
   :preconds '( (is_at AG home)
                (is_tired_to_degree AG ?t)
                (>= ?f 0.5)
                (is_hungry_to_degree AG ?h)
                (> ?f ?h)
                (not (there_is_a_fire)) )
   :effects '( (is_tired_to_degree AG 0)
               (not (is_tired_to_degree AG ?f))
```

```
                (is_hungry_to_degree AG (+ ?h 2)))
   :time-required '(* 4 ?f)
   :value '(* 2 ?f)
   )
)
```

From AG's perspective, if it is at home, is more tired than hungry, is at least of fatigue level 0.5, and there is no fire, then it can sleep for a duration given by $(* \ 4 \ ?f)$ and, as a result, it will relieve its fatigue at the expense of increasing its hunger level by 2. Performing an instantiated (i.e., binding specific input arguments to its formal parameters) *sleep* action will afford AG a net increase of $(* \ 2 \ ?f)$ in its cumulative utility.

Since *AG*'s knowledge of the world is incomplete, the actual effects of its actions may diverge from the effects expected by the agent. For example, if we model traveling (from *AG*'s perspective) as a multi-step action, but also allow for spontaneous fires that bring travel to a halt, then *AG* may not reach its expected destination. Given this divergence between expectations and reality, we clearly need to distinguish between *AG*'s conception of its actions and the *actual* actions in the simulated world. Therefore, for each model operator you define, you will need to define its actual version (named with suffix *.actual*) to model the corresponding actual action in the simulated world separately from *AG*'s conception of that action.

Consider the stepwise, *actual* version *sleep.actual* of the *sleep* operator:

```
(setq sleep.actual
   (make-op.actual
   :name 'sleep.actual
   :pars '(?f ?h)
   :startconds '( (is_at AG home)
                  (is_tired_to_degree AG ?t)
                  (>= ?f 0.5)
                  (is_hungry_to_degree AG ?h)
                  (> ?f ?h) )
   :stopconds '( (there_is_a_fire)
                 (is_tired_to_degree AG 0) )
   :deletes '( (is_tired_to_degree AG ?#1)
               (is_hungry_to_degree AG ?#2) )
   :adds '( (is_tired_to_degree AG (- ?f (* 0.5 (elapsed_time?))))
            (is_hungry_to_degree AG (+ ?h (* 0.5 (elapsed_time?)))) )
   )
)
```

Notably, the actual version does not have the anticipated value or time-required field. The simulation assumes that values of actions specified in the model operators are correct, and so don't have to be specified in actual operators. Once AG chooses to execute a certain action, AG's utility is calculated by summing the anticipated value from the action's model and the actual new state reached. Actual utilities for AG are calculated

and maintained this way. The start conditions as given by *startconds* are the same but for the removal of the (*not* (*there_is_a_fire*)) formula. Now the action will continue for another time step if and only if neither of its stop conditions as given by *stopconds* is true in the current state. If at least one of them is true in the current state, then the action will immediately terminate. Otherwise, the current state and AG's knowledge base will be updated with AG's lower fatigue level and higher hunger level; the update is done by first removing the literals of *deletes* and then adding the literals of *adds*.

Plans will consist of sequences of actions with particular values for the parameters. The idea is for *AG* to maintain a plan at all times, adding new goals and new steps as it moves around and acts in the Gridworld. Its reasons for adding goals, and more generally for modifying its current plan, have to do with its preferences, i.e., how it *evaluates* various states and actions. Planning is done by forward-search from a given state, followed my propagating backward the rewards and costs of the various actions and states reached, to obtain best-possible values and hence the seemingly best sequence of actions.

The basic forward search function is **chain-forward**, and this is called upon by the **go!** function; they are in files **gridworld-planning.lisp** and **simulation-and-function-evaluation.lisp**, respectively. The does a systematic forward search constrained by a *search beam*, then actually executes the seemingly best next action, and updates the state and its knowledge with the action effects, its new observations of non-occluded local facts, and its new general-knowledge inferences (see Section 2.5 for details).

The forward search is bounded by a prespecified *search beam*, which specifies the number of lookahead levels (the length of the contemplated sequences of actions), the branching factor and the allowable operators at each level. You should set your search beam accordingly for the Gridworld that you design, by appropriate assignment to the global variables ***operators*** and ***search-beam*** in your source file. For example, the following sets the search beam to a three-level lookahead, with branching factors 5, 4, and 3 for the first, second, and third levels, respectively, with the same set of allowable operators to consider at each level (though you can certainly specify a different set of allowable operators to consider for each level of the lookahead):

```
(setq *operators* '(walk eat answer_user_ynq answer_user_whq sleep drink))
(setq *search-beam*
   (list (cons 5 *operators*) (cons 4 *operators*) (cons 3 *operators*)))
```

In addition to setting the search beam for the Gridworld that you design, you need to specify the rewards and costs of the various actions and potential states reached, as follows. First, you should prescribe the anticipated net utility of each model operator you design and use in your Gridworld by setting the : *value* field of each model operator accordingly; this value could either be a static numerical measure or be dependent on (i.e., be a function of) the specific input arguments that would be substituted into the formal parameters in instantiating the model operator into a contemplated action. Second, for the Gridworld that you design, you should also assign values to specific properties that could be found in a state by modifying the contents of LISP routine **(defun state-value (additions deletions prior-local-value) ...)** in file **gridworld-planning.lisp**. This

function essentially allocates positive points for addition of desirable properties to a state, and subtracts away these points for deletions of those properties; analogously for addition of undesirable properties to, and removal of undesirable properties from a state. To see an example, please refer to the header comments of this LISP routine.

2.4.1 Procedural Attachment

The STRIPS-like action representation allows for quantitative preconditions and effects, handled by a general procedural attachment syntax. The uniform procedural attachment technique is used in preconditions and effects (and value and duration, if applicable) of both the model actions and the actual actions, and it enables both quantitative reasoning and dialogue. You may define action preconditions and effects (and value and duration, if applicable) containing *evaluable terms*, and these are the functions or predicates whose operators are among $+$, $-$, $*$, $/$, $<$, $<=$, $=$, $>=$, $>$, *random*, and user-defined (i.e., defined by you) function or predicate names ending in ?.

The system will evaluate the evaluable terms when verifying preconditions and when applying effects (and when contemplating the net utility and duration, if applicable). For example, $(is\_tired\_to\_degree\ AG\ (+\ ?f\ (*\ 0.5\ (elapsed\_time?))))$, an effect of *walk.actual*, specifies that the increase in *AG*'s fatigue level as a result of the walking action will be half the distance it walks, where the user-defined function $(elapsed\_time?)$ returns the time *AG* has spent on the current walking action assuming *AG* walks 1 distance unit per time unit in the simulated world.

Not only does procedural attachment handle quantitative preconditions and effects (and value and duration, if applicable) effectively, but it also handles side-effects such as *AG* producing a printed answer straightforwardly. For instance, when applying the effect $(knows\ USER\ (that\ (answer\_to\_ynq?\ ?q)))$ of operator *answer_ynq* to *AG*'s knowledge base, the evaluable term $(answer\_to\_ynq?\ ?q)$ is evaluated as an answer formula; the answer formula might for example be $(not\ (can\_fly\ guru))$ for $?q$ being $(can\_fly\ guru)$. In other words, $(answer\_to\_ynq?\ ?q)$ is evaluated as an answer formula by applying the LISP function *answer_to_ynq*? to the value bound to $?q$, where function *answer_to_ynq*? makes use of *AG*'s knowledge base to determine an answer.

2.5 Reasoning about World States and Mental States

Since *AG* does not generally know all the current facts, it cannot make full use of the *closed world assumption (CWA)* (unlike many traditional planners). But we do not want to abandon the CWA altogether, as it would be inefficient to enumerate and maintain all the negative predications that hold even in simple worlds. Thus, *AG* uses the full CWA only for (non-epistemic) literals in which *AG* is the subject of the literal. In this respect its self-knowledge is assumed to be complete.

But when the literal concerns a non-*AG* subject, *AG* applies the CWA only for the two following cases:

1. literals about road connectivity and navigability (even here the CWA could easily be weakened); e.g., the absence of $(road\ path5)$ from *AG*'s knowledge base, re-

gardless of whether (*not* (*road path*5)) is present, would suffice for *AG* to conclude that *path*5 is not a *road*;

2. when the subject is a local entity currently colocated with *AG* or one *AG* has visited, and the predicate is non-occluded (defined below).

In all other cases concerning a non-*AG* subject, the mere absence of a literal from the world model is not interpreted as supporting its negation – its truth value may simply be unknown. Note that *AG* may learn that (*not* ϕ) holds for some literal ϕ even if ϕ is not subject to the CWA; e.g., after opening some box *box*1 to find out about its previously occluded contents, *AG* may learn that (*not* (*is_in key*1 *box*1)). In such a case, it stores that negative literal in its world model. Therefore, in checking whether a literal ϕ is true or false, where ϕ is not subject to the CWA, *AG* will not conclude "unknown" merely because ϕ is absent from its world model – it also checks whether the negation is explicitly known. *AG* updates its knowledge of the non-occluded properties of an entity only when it (re)visits its location, and its knowledge of the occluded properties of that entity when it takes appropriate actions to (re)discover them. The method of judging the truth value of a non-epistemic ground atom is summarized more formally in Algorithm 1. A negative literal is judged to be true, false, or unknown accordingly as the atom that it negates is judged to be false, true, or unknown.

Occluded predicates are those predicates for local facts that we want to regard as not immediately known to *AG*, unless the first (subject) argument is *AG*. For example, the predicate *is_in* might be regarded as occluded, so if (*is_in key*1 *box*1) holds, *AG* does not know this even when standing next to *box*1. Similarly *knows* would generally be occluded, so that what another agent knows is not immediately apparent to *AG*. However, self-referential facts about *AG* such as (*has AG key*1) and (*not* (*knows AG* (*whether* (*is_edible fruit*3)))) are regarded as evident to *AG* (and thus added to *AG*'s world model), despite the general occlusion of *has* and *knows*. Also, *AG* may find out and remember a fact that would otherwise be occluded, perhaps because it asked a question, read a note containing the fact, or inferred it. To mark certain predicates as being occluded, you will need to replace and set the value of the global variable ***occluded-preds*** to your list of occluded predicates, via the line **(defvar *occluded-preds* ...)** in file **gridworld-definitions.lisp**. Predicates not explicitly marked by you as being occluded are non-occluded by default.

In the case of epistemic predications, *AG* judges their truth value by an introspection algorithm rather than closed-world inference. In evaluating a predication of form (*knows SUBJ* (*that* ψ)) (e.g., (*knows AG* (*that* (*can_talk guru*)))) with ψ being a literal, the algorithm considers the two cases $SUBJ = AG$ and $SUBJ \neq AG$. In the first case, *AG* uses the methods in the previous paragraph to determine whether ψ is true, false or unknown, and judges the autoepistemic predication to be true, false or false respectively (i.e., in the latter two cases, *AG* does not know ψ to be true). In the case $SUBJ \neq AG$, *AG* judges the epistemic predication true only if *SUBJ* is identical to the subject of ψ (thus making a similar knowledge assumption about other agents as for itself, and false otherwise (a negative closure assumption that could be weakened). The method for predications of form (*knows SUBJ* (*whether* ψ)) is much the same. But in

the case $SUBJ = AG$, when ψ is found to be true, false or unknown, the autoepistemic predication is judged to be true, true or false respectively. In the case $SUBJ \neq AG$, AG again judges the epistemic predication as true only if $SUBJ$ is identical to the subject of ψ, and false otherwise. The method of evaluating epistemic predications is depicted in Algorithm 2.

Algorithm 1 Evaluating Non-epistemic Atoms Using a Restricted CWA

1: Let (P t_1 t_2 ...) be a nonepistemic ground atom.
{We assume that t_1, t_2, ... are either constants, or *evaluable terms* which have been evaluated to constants.}
{Either t_1, t_2, ... are constants (in which case even a *knows*-predication such as (*knows AG Guru*) is allowable here), or else P is a predicate other than *knows* (but we allow, e.g., *wants*, *wonders*, or *contains_a_message*), and t_1, t_2, ... are variable-free terms (constants, or reified propositions formed by applying *that* to a ground literal, or reified yes-no questions formed by applying *whether* to a ground literal).}
{We judge the truth value of (P t_1 t_2 ...) as *true*, *false*, or *unknown* in a given state (as modeled by AG) according to the following rules.}
2: **if** (P t_1 t_2 ...) is in the state description **then**
3: **return** *true*
4: **end if**
5: **if** (*not* (P t_1 t_2 ...)) is in the state description **then**
6: **return** *false*
7: **end if**
{To reach this point in the algorithm must mean that neither (P t_1 t_2 ...) nor (*not* (P t_1 t_2 ...)) is in the state description.}
8: **if** P is one of *road*, *connects*, *navigable*, ..., i.e., a roadmap predicate **then**
9: **return** *false*
10: **else if** $t_1 = AG$ **then**
11: **return** *false*
12: **end if**
13: **if** P is not marked as an occluded predicate, and t_1 denotes a local entity (colocated with AG) or an entity local to a place that AG has visited **then**
14: **return** *false*
15: **end if**
16: **return** *unknown*

AG also performs bounded forward inference for any state that it reaches in its simulated world or in its lookahead, based on all of its current factual knowledge and all of its general quantified knowledge. For example, from (*knows guru* (*that p*)) AG can infer both p and (*knows guru* (*whether p*)); from (*sasquatch moe*) and general knowledge ((*sasquatch* ?x) => (*has_IQ* ?x 50)) where variable ?x is assumed to be universally quantified over the domain, AG can infer that (*has_IQ moe* 50). Currently, the depth of bounded forward inference is set at 2, but you can certainly reset this by replacing

Algorithm 2 Evaluating Epistemic Atoms Using Introspection

Let ϕ be of the form $(knows\ t_1\ (that\ \psi))$ or $(knows\ t_1\ (whether\ \psi))$.
{t_1 is a constant, and ψ is a ground, possibly epistemic literal.}
{We assume that any embedded terms such as $(+\ account-balance?\ 100)$ have been evaluated to constants.}
{We judge the truth of ϕ as $true$, $false$, or $unknown$ in a given state (as modeled by AG) according to the following rules.}
if ϕ is in the state description **then**
  return $true$
end if
if $(not\ \phi)$ is in the state description **then**
  return $false$
end if
{To reach this point in the algorithm must mean that neither ϕ nor $(not\ \phi)$ is in the state description.}
Evaluate ψ to $true$, $false$, or $unknown$ recursively using the method of Algorithm 1 or the method of Algorithm 2, depending on whether ψ is non-epistemic or epistemic.
if $t_1 = AG$, or the subject of ψ (the first argument appearing in this literal) is also t_1 **then**
  if ϕ is of the form $(knows\ t_1\ (that\ \psi))$ **then**
    if value of ψ is $unknown$ **then**
      return $false$
    else
      {Value of ψ is either $true$ or $false$.}
      return value of ψ
    end if
  else
    {ϕ is of the form $(knows\ t_1\ (whether\ \psi))$}
    if value of ψ is $unknown$ **then**
      return $false$
    else
      {Value of ψ is either $true$ or $false$.}
      return $true$
    end if
  end if
else
  return $false$
end if

the value in the line **(defvar *inference-limit* ...)** in file **gridworld-planning.lisp**.

For the Gridworld that you design, you can also manually add inference rules to your agent's general knowledge global variable ***general-knowledge*** in your source file, using the line $(push\ (list\ \phi\ \texttt{'=>}\ \psi)\ *general-knowledge*)$
where antecedent ϕ is either a single positive literal or a conjunction of positive literals (i.e., of the form $(and\ \phi_1\ \phi_2\ ...\ \phi_k)$), consequent ψ is a positive literal, and `'=>` are three consecutive characters/symbols (namely, a single quote, an equal sign, and a greater-than sign). The general knowledge inference rules implemented in the framework are more general than standard Horn clauses as they allow for propositional attitude predications.

As previously stated, the logic of state descriptions allows for propositional attitudes such as *knowing-that*, *knowing-whether* and *wanting*. This is essential for formalizing knowledge-acquisition actions and for answering questions about the agent's own attitudes. To be syntactically correct, the object arguments of such (epistemic or not) predicates as *wants* and *knows* must be reified with $(whether\ (...))$ or $(that\ (...))$, if they are literals themselves. For instance, $(wants\ AG\ (that\ (drinks\ AG\ juice)))$ expresses that *AG* wants to drink *juice*, and $(knows\ AG\ (whether\ (is\_alive\ AG)))$ states that *AG* knows whether *AG* is alive. On the other hand, the object argument of $(wants\ AG\ juice)$ and $(knows\ AG\ juice)$ need not be reified with $(whether(...))$ or $(that(...))$ since *juice* is not a literal.

In summary, *AG*'s inference capabilities, like those detailed in this section, are especially important in two respects. First, they are instrumental in *AG*'s attempt to confirm or disconfirm action preconditions, including knowledge preconditions in actions like asking an agent whether p is true (viz., not knowing whether p is true, but believing that the agent knows whether p is true). Second, the inference capabilities are crucial in answering questions, including ones that test *AG*'s introspective capabilities. Given a question, *AG* will either inform the user if it doesn't know the answer, or otherwise verbalize its answer(s) as English sentence(s).

2.6 Running your own Gridworld

Once you have designed and coded your own Gridworld, you can run it to experiment with it.

1. Open a terminal window and make sure you change directory (cd) into the one containing the LISP source files. Type and enter "acl" (without quotes) at the command prompt to start Allegro Common LISP installed on the department machines.

2. Load the four files in this order: gridworld-definitions.lisp, gridworld-planning.lisp, simulation-and-function-evaluation.lisp, and lastly your own Gridworld LISP source file (e.g., your-own-world.lisp).

```
(load "gridworld-definitions.lisp")
(load "gridworld-planning.lisp")
(load "simulation-and-function-evaluation.lisp")
```

```
(load "your-own-world.lisp")
```

3. Type and enter at the prompt (*initialize−state−node*) to initialize your loaded Gridworld

4. Enter (*go*!) to begin/continue your agent's exploration, or (*listen*!) to input wh- and yes/no-questions to your agent. Repeat this step until you want to stop running your agent in your Gridworld.

5. Type and enter (*exit*) to exit Allegro Common LISP.

3 Experimental Agent

We implemented a version of *AG* in our framework to demonstrate self-motivated question-answering, and *AG*'s opportunistic behavior resulting from its continual, deliberate, reward-seeking planning. This Gridworld example is implemented in the provided source file **gridworld-world.lisp**. To run our Gridworld example, follow the instructions in Section 2.6 but replace "your-own-world.lisp" with **"gridworld-world.lisp"**.

3.1 Simulated World

We situate *AG* in a very simple simulated world, yet one sufficiently subtle to illustrate the above features. This world is inhabited by animate agents *AG* and *guru*, along with inanimate objects *pizza*, *juice*, and *piano*. There are three locations *home*, *grove*, and *plaza*, with roads *path*1 and *path*2 as shown in Figure 1. Object *pizza* is edible and at *home*; *juice* is potable and at *plaza*; *piano* is playable and at *home*. For demonstration purposes, *pizza* and *juice* are presumed inexhaustible. Agent *guru* likes *piano* and knows whether *juice* is potable.

Initially *AG* is at home, not tired, and has a hunger level of 4.0 and a thirst level of 2.0. In addition to knowledge of the road network, the existence of the objects at *home* and (via forward inference) their general type properties, *AG* knows that *juice* is at *plaza*, and that *guru* located at *grove* can talk and knows whether *juice* is potable. *AG* has the following operators at its disposal (where associated reward levels are indicated parenthetically): *eat* (2 * *AG*'s hunger level at the onset of the action), *drink* (2 * *AG*'s thirst level at the onset of the action), *walk* (3 - *AG*'s fatigue level at the onset of the action), *sleep* (*AG*'s fatigue level at the onset of the action), *play* (3), *ask other agents whether something is true* (5), and *answer the user's yes/no and wh-questions* (10).

Any state *s* reached by *AG* in its projective lookahead has an inherent value, based on the following criteria comparing *s* with its parent state. Answering a user question yields 50 utility points; becoming not bored gains *AG* 1 point; becoming more tired (thirstier, hungrier) costs *AG* points equal to the increase in *AG*'s fatigue (thirst, hunger) level. Conversely, leaving a user question unanswered costs *AG* 20 points; becoming bored costs *AG* 1 point; becoming less tired (thirsty, hungry) gains *AG* points equal to the decrease in *AG*'s fatigue (thirst, hunger) level. These criteria, along with the operators' anticipated rewards, suggest that *AG* likes answering user's questions the most.

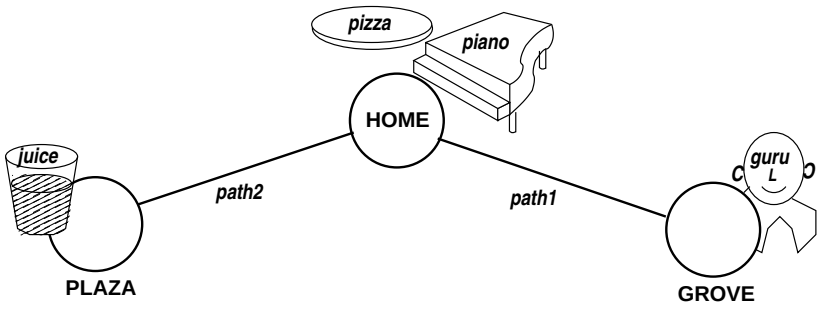

Figure 1: Our Simulated World

A potentially multi-step action faces the possibility of interference by two types of exogenous events, namely rain and fire, but only fire may disrupt the action (e.g., sleeping, traveling). Spontaneous rain has a 33% chance of starting; once begun, it has a 25% chance of stopping. As long as there is no rain, a spontaneous fire has a 5% chance of starting; once started, it has a 50% chance of dying by itself, and also goes out when there is rain.

3.2 Question-Answering

AG's dialogue interaction is handled uniformly via *AG*'s planning and procedural attachment capabilities. Input questions currently must be expressed in the same syntax as that for symbolically representing knowledge in the simulated world. A yes/no question is prefixed with *ask-yn*; a wh-question, with *ask-wh*. Though more than one question can be entered at a time, *AG* handles questions independently, and it may be some time until *AG* chooses to answer a question as the next best action to perform. To pose questions to *AG*, the user types and enters command (*listen*!) at the prompt, and then inputs the questions. For more information, please refer to the header comments of LISP routine **listen!** in file **simulation-and-function-evaluation.lisp**.

The following example is self-explanatory.

```
>> (listen!)
((ask-yn (pizza is_tasty))
 (ask-yn (not (AG is_bored))))

ACTION: (ANSWER_YNQ (IS_TASTY PIZZA))
Answer: (AG DOES NOT KNOW WHETHER PIZZA IS TASTY)
 For question (IS_TASTY PIZZA), AG doesn't know
 the answer as it may be about a non-local object
 or about an occluded property of a local object.

ACTION: (ANSWER_YNQ (NOT (IS_BORED AG)))
Answer: (IT IS NOT THE CASE THAT AG IS BORED)
 For question (NOT (IS_BORED AG)), AG offers
 the answer above based on its current knowledge.
```

An example of a yes/no question that would currently be hard to answer correctly is (*AG is_hungry*). This is tricky for two reasons. First, currently in the framework and also in the experimental Gridworld of **gridworld-world.lisp**, the inference rule that "if *AG*'s hunger level is greater than 0, then *AG* is hungry" has not been created and added. Second, all hunger-related predications currently have the form (*is_hungry_to_degree* AG *n*) where *n* is a number or what would evaluate to a number. Hence, *AG* would currently always answer "it is not the case that *AG* is hungry" given that (*is_hungry AG*) has an unknown truth value and (by virtue of being animate) *AG* should itself already know whether it is hungry. The (auto)epistemic reasoning to arrive at this answer is explained in Section 2.5. This illustrates the importance of *AG*'s being

able to make inferences using the general knowledge in the global variable ***general-knowledge***, so you may also need to augment your own Gridworld with appropriate inference rules to make question-answering sensible.

A wh-question of the form (*ask-wh* r) must have at least one variable (indicated by prefix ?) in r. For instance, (*not* (*AG likes* ?x)) corresponds to the question "what does *AG* not like?" while (?y *is_at* ?z) translates to "where is every entity located?". In computing the answer(s), *AG* attempts to unify r with facts in its current knowledge base; for each set s of bindings found for the variables in r, *AG* forms the corresponding answer by replacing variables in r with bindings in s. *AG* uses the weakened CWA to verbalize its responses as follows.

```
>> (listen!)
((ask-wh (AG is_tired_to_degree ?x))
 (ask-wh (?y is_animate))
 (ask-wh (?z is_bored)))

ACTION: (ANSWER_WHQ (IS_TIRED_TO_DEGREE AG ?X))
ANSWER: (AG IS TIRED TO DEGREE 1.5)
 For question (NOT (IS_TIRED_TO_DEGREE AG ?X)),
 other than the above positive instance(s), AG
 assumes everything else as the answer.

ACTION: (ANSWER_WHQ (IS_ANIMATE ?Y))
ANSWER: (AG IS ANIMATE) (GURU IS ANIMATE)
 For question (IS_ANIMATE ?Y), other than the
 above positive instance(s), AG doesn't know
 anything else as the answer.

ACTION: (ANSWER_WHQ (IS_BORED ?Z))
ANSWER: (AG DOES NOT KNOW WHETHER ANYTHING IS BORED)
 For question (IS_BORED ?Z), AG knows of no
 positive instances, so AG doesn't know anything
 as the answer.
```

3.3 Example Results

We give further examples of *AG*'s question-answering, and empirical results showing *AG*'s opportunistic behavior (due to its self-motivated, deliberate, continual planning) in the context of the simulated world.

The following, including knows-whether and knows-that questions, is a concatenation of several dialogue exchanges between *AG* and the user, showing only *AG*'s actions to answer user questions and *AG*'s corresponding verbalized English answers while omitting system output. The examples showcase *AG*'s ability to introspect positively and negatively using the relaxed CWA.

```
ACTION: (ANSWER_YNQ (KNOWS GURU (WHETHER (LIKES GURU PIZZA))))
ANSWER: (GURU KNOWS WHETHER GURU LIKES PIZZA)

ACTION: (ANSWER_YNQ (KNOWS GURU (WHETHER (LIKES AG PIZZA))))
ANSWER: (AG DOES NOT KNOW WHETHER GURU KNOWS WHETHER AG LIKES PIZZA)

ACTION: (ANSWER_YNQ (CAN_FLY GURU))
ANSWER: (IT IS NOT THE CASE THAT GURU CAN FLY)

ACTION: (ANSWER_YNQ (KNOWS GURU (THAT (CAN_FLY GURU))))
ANSWER: (GURU KNOWS THAT IT IS NOT THE CASE THAT GURU CAN FLY)
```

We now shift our attention to some experimental results wherein *AG*'s behavior shows both opportunism and foresight. The success of a run is measured in terms of the *net utility* (NU = cumulative rewards - cumulative penalties), summed over the entire sequence of actions and corresponding states. The following run shows *AG*'s sequence of 10 actions, using a three-level lookahead with branching factors of 5, 4, and 3, successively. This sequence was stored in the global variable ***AG-history***, and its NU was 60.5 as stored in the global variable ***total-value***; you can read and display the contents of both these global variables to the screen. Each action was logged along with its first iteration, anticipated duration, and time in the simulated world when that iteration began; also, each multi-step action was logged again with its final iteration.

```
((EAT 4.0 PIZZA HOME) 1 1 0)
((WALK HOME GROVE PATH1 0.0) 1 3 2)
((WALK HOME GROVE PATH1 0.0) 3 3 4)
((ASK+WHETHER GURU (POTABLE JUICE) GROVE) 1 1 6)
((WALK GROVE HOME PATH1 1.0) 1 3 8)
((WALK GROVE HOME PATH1 1.0) 3 3 10)
((SLEEP 2.0 0.0) 1 12.0 12)
((SLEEP 2.0 0.0) 6 12.0 17)
((WALK HOME PLAZA PATH2 0.0) 1 2 19)
((WALK HOME PLAZA PATH2 0.0) 2 2 20)
((DRINK 2.0 JUICE PLAZA) 1 1 22)
((WALK PLAZA HOME PATH2 1.0) 1 2 24)
((WALK PLAZA HOME PATH2 1.0) 2 2 25)
((WALK HOME PLAZA PATH2 0.0) 1 2 27)
((WALK HOME PLAZA PATH2 0.0) 2 2 28)
((WALK PLAZA HOME PATH2 1.0) 1 2 30)
((WALK PLAZA HOME PATH2 1.0) 2 2 31)
```

AG's chosen seemingly best action in itself alone may not be immediately rewarding to *AG*, but rather is anticipated by *AG* to lead to a most rewarding sequence of actions. For example, though *AG* might not get a reward for walking from *home* to *grove*, it may

very well foresee a high reward resulting from traveling to *grove* to meet *guru*, asking *guru* to learn about *juice*'s potability (knowledge which *AG* initially does not have), traveling to *plaza* where *juice* is, and eventually drinking *juice*. Thus, the use of the reasoned, projective lookahead enables *AG* to exhibit foresight and opportunism in its behavior.

4 Coding Guidelines

Here are some guidelines you should follow when designing and coding your own Gridworld.

1. Although the Gridworld Framework is amenable to goal-directed planning (with a large enough search beam in lookahead) as has been shown in our publications, keep in mind that for the purpose of the Consciousness course, you are to design a reasonable agent that shows *self-motivation* in its planning and execution of actions. You are not striving for a dutiful goal-directed planner.

2. Follow the usual good programming style, giving all your functions, predicates, variables, and action operator definitions meaningful names and also abiding by their naming conventions and syntactical constructs as instructed in the preceding sections.

3. Use originality in creating your own Gridworld. That said, you are welcome to reuse some action operator definitions that we have used in gridworld-world.lisp. However, explicitly state which of our operator definitions you are reusing and/or have modified. Also, do not copy and paste all of our operator definitions into your own code as that would make for less understandable code; rather, include only the ones you are reusing in your code.

4. Comment your code thoroughly.
   - At the beginning of each function and each action operator definition, provide a intuitive statement of its purpose, the meaning and kind of value of each of the parameters (with an example if the value has some special form), and the output or effect of the function.
   - If anything is unclear in the body of a function, explain what is going on. This ranges from places where you are taking cars of cdrs of cars, to places where your recursive solution hinges on the fact that some recursive call has already taken care of the special case that the user might wonder about. When in doubt, comment.

Your project grade will depend on how well you demonstrate originality in your Gridworld, adhere to coding guidelines, and meet requirements set out by the instructor for the project. Requirements for your test files and report (for submission) will be distributed in class.

5 Required Readings

To acquaint yourself with the theoretical underpinnings and more examples of the Grid-world framework, read the two following publications available through Prof. Len Schubert's department Webpage (`http://www.cs.rochester.edu/u/schubert/`):

1. Daphne Liu and Lenhart Schubert. Combining Self-Motivation with Logical Planning and Inference in a Reward-Seeking Agent. Paper for poster, in Proceedings of the Second International Conference on Agents and Artificial Intelligence (ICAART 2010), vol. 2 (INSTICC Press), Valencia, Spain, January 22-24, 2010, pp. 257-263.

2. Daphne Liu and Lenhart Schubert. Incorporating Planning and Reasoning into a Self-Motivated, Communicative Agent. In Proceedings of the Second Conference on Artificial General Intelligence (AGI 2009), Arlington, VA, March 6-9, 2009, pp. 108-113.