

# Preliminary APE Manual

10/17/2006

## 1. Introduction

The purpose of this document is to describe the APE system: how it works, how to use it, and how to interface with it.

APE can be used to implement any interactive dialogue system, where turns in the “conversation” may include graphical actions and/or text. APE is intended largely to model dialogues with hierarchical, multi-turn plans, especially dialogues involving multiple types of answers to a question, each of which requires a different continuation.

However, it can be used as a general-purpose programming tool for implementing a dialogue system, since it can express the fundamental building blocks of sequence, selection and iteration (Dijkstra, 1972).

## 2. Internals

### 2.1 Basic loop, including unification

The system stores its intentions in an *agenda*, which is implemented as a stack with additional operations available besides *push* and *pop*. The agenda contains the stack of unsatisfied goals. Agenda entries also include the current plan operator for each goal and its bindings.

To initiate a planning session, the user invokes the planner with a goal, e.g., to conduct a conversation on a specified topic or with a specific purpose. The system stores the initial goal on the agenda, then searches the operator library to find all operators whose *goal* field matches the goal on top of the agenda and whose *filter conditions* and *preconditions* are satisfied. The filter slot is used for static properties, such as properties of domain objects, while preconditions are used for characteristics that can change as the dialogue progresses, e.g. the number of times a particular construct has been used.

The filter slot is used for efficiency only; all preconditions can be coded in the precondition slot. (For simplicity, the filter slot is not shown in the examples.)

Goals are represented using first-order logic without quantifiers, with unification used for matching. Multiple conditions can be used, and are considered to be *and*-ed together. The *not* function is also available; the closed world assumption is used. If more than one ⟨operator, binding list⟩ match is found, the last one found is used, although the user has the option of writing a different algorithm for resolving duplicates.

During unification, repeated instances of a variable within a rule are considered identical. However, since each rule is considered individually, instances of a variables in different rules are considered distinct. To keep a binding made while evaluating a rule, store the information in the transient knowledge base.

Since the system is intended especially for generation of hierarchically organized task-oriented

discourse, each operator has a multi-step *recipe* in the style of Wilkins (1988). When a match is found, the matching goal is removed from the agenda and is replaced by the following items (top of list represents top of agenda):

- `<BEGIN>` marker
- First step of recipe
- Second step of recipe
- ...
- `<END>` marker

The `<END>` marker contains the goal that triggered the operator in case we later need to find another way to satisfy the same goal.

## 2.2 External data structures

The APE environment contains two knowledge bases of ground clauses. One contains permanent information such as domain facts, while the other contains transient facts that become *true* during program execution.

Facts can be added and deleted from the dynamic knowledge base whenever desired using *assert* and *retract*. Additionally, the *:hiercx* slot of the operator syntax exists to allow operators to make decisions according to the current discourse context. Items in the *:hiercx* slot are instantiated and kept in the dynamic knowledge base only as long as the operator which spawned them is on the agenda.

Finally, APE permits the user to declare *external relations*. These relations are used in the same way as other knowledge base relations, but they do not actually appear in the knowledge base. Instead, when an external relation is called, a user-written function is requested to return a binding list with all possible sets of bindings, in the same format used by the knowledge base interface. The use of external relations allows the user to express preconditions which may be difficult or impossible to express in first-order logic, such as arithmetic relations.

External relations can also be used to communicate with another program, such as a GUI, a database system, an automated reasoning system, or a domain expert. In the examples below, names starting with *e-* represent external relations.

## 3. Operator syntax and semantics

### 3.1 Syntax

- *Goal*: goal of operator
- *Filter*: a list of logical expressions that must be true in order to consider running the operator
- *Precond*: a list of logical expressions that must be true in order to run the operator
- *Recipe*: list of recipe steps for operator
- *Hiercx*: knowledge base forms which are true while operator is in play

The *:hiercx* slot of the operator exists to allow operators to make decisions according to the

current discourse context. Items in the *:hiercx* slot are instantiated and kept in the dynamic knowledge base only as long as the operator which spawned them is on the agenda.

### 3.2 Recipe item types

Recipe items can take several forms:

- *Goal*: Create a subgoal.
- *Primitive*: Do an action. Since this is a text planner, the action is usually to say something.

Graphical actions can be implemented as primitives or via the use of an external relation.

- *Interactive primitive*: Say something and give control to the other party to reply.
- *Assert*: Add a fact to the dynamic knowledge base.
- *Retract*: Remove all matching facts from the dynamic knowledge base.
- *Fact*: Evaluate a condition. If false, skip the rest of the recipe.

*Fact* is used to allow run-time decision making by bypassing the rest of an operator when circumstances change during its execution. Although preconditions and *fact* both allow one to express *if-then* conditions in the APE syntax, preconditions are only checked when an operator is being chosen. Note that *fact* does not create new bindings. If the bindings that make a *fact* true are needed, use a goal after the fact as in the example below. (Although the use of *fact* could be replaced by preconditions on an additional operator, the use of *fact* allows a more natural writing style.)

```
(def-operator do-one-student
  :goal (did-one-student)
  :filter ()
  :precond ()
  :recipe ((goal (did-obtain student-name))
           (fact (w-student-name-is ?student))           ;; i.e. not menu-item
           (goal (w-student-name-is ?student))           ;; to get binding
           (goal (did-student ?student)))
  :hiercx ())
```

- *Retry-at*: Pop the agenda through the first  $\langle$ END $\rangle$  marker where the retry argument is false, then restore that entry's original goal.

*Retry-at* implements a Prolog-like choice of alternatives. If there are multiple ways to satisfy a goal, *retry-at* allows one to choose among them a second time if one's first choice is later shown to be undesirable. For *retry-at* to be useful, the author must provide multiple operators for the same goal. Each operator must have a set of preconditions enabling it to be chosen at the appropriate time. As in Prolog, *retry-at* is also useful for implementing loops; one simply chooses the same operator every time until the exit condition is reached. *Fact* can be used to check the exit condition.

- *Prune-replace*: Pop the agenda until the retry argument becomes false, then push an optional list of new recipe items onto the agenda.

*Prune-replace* allows a type of decision-making frequently used in dialogue generation. When a conversation partner does not give the expected response, one would often like to pop the top

goal from the agenda and replace it with one or more replacement goals. *Prune-replace* implements a generalized version of this concept, allowing one to pop the agenda until a desired configuration is reached. (Although any instance of *prune-replace* could be replaced by *retry-at* and an additional operator, the use of *prune-replace* permits a more natural writing style.)

### 3.3 Naming conventions

Names beginning with *ipe-* are reserved for system use.

Variable names can't end with a digit, as the system uniquifies variables by adding a numerical suffix.

The term *nil* is reserved for system use (with its standard meaning of *false*) and therefore cannot be used as an argument to relations.

### 3.4 Naming suggestions

Although APE puts no restriction on relation names, a naming convention similar to the following is suggested in order to keep track of relations. In the examples below, the following naming convention is used:

Relations starting with *w-* are “working storage,” i.e. facts added to the dynamic knowledge base to maintain state between turns.

Relations starting with *e-* represent external relations.

Relations starting with *i-* are used to convey information from the user, i.e. they are added to the knowledge base by the user interface.

Many relations in the examples below use the suffix *-is* to indicate that an object is a relation rather than a function.

Similarly, most of the goals use the prefix *did-* to indicate that the goal will be satisfied with the accomplishment of an event.

## 4. Options

The following options may be useful for debugging.

:agenda	display agenda each time through loop
:global	print global bindings each time through loop
:op	display chosen operators and stack edits
:mini-op	display only stack edits
:long-op	show alternate operators
:binding	display binding events
:kb	display KB assert & retract
:conj	display details of conjunction (multiple precondition) processing
:bag	display details of bag processing

The following options are useful in restricted circumstances:

:plan-history	display SGML-like listing
:primitives	display only S and T turns, no higher level plans
:plain	omit head of primitive (e.g. ask, say) in display
:say-errors	determine whether tutor should mention missing operators
:no-bc	don't do backward chaining
:long-agenda	print full record for <end> records (default is just for <begin> records)

## 5. External functions

### 5.1 Using external functions

External functions must be identified at load time. External functions belonging to the APE system are listed in the APE load file `loader.lsp`. Users can add additional functions in their own load files. Without some way to identify an external function, the system cannot tell an external function from a KB predicate with no entries. Thus an external function not identified as such will always return *false*.

Note that some external functions only work with certain combinations of arguments instantiated as input.

### 5.2 Examining the goal stack

*e-current-goal-is* returns the top goal on the agenda. If the argument is instantiated, it returns *true* if the argument is on top of the stack and fails otherwise. If the argument is not instantiated, it binds the argument to the goal at the top of the agenda. If the top of the stack is not a goal, it fails.

*e-current-goal-pred-is* returns the predicate of the top goal on the agenda. If the argument is instantiated, it returns *true* if the argument is on top of the agenda and *false* otherwise. If the argument is not instantiated, it binds the argument to the predicate of the goal at the top of the agenda. If the top of the agenda is not a goal, it fails. This function is useful because we usually don't care what the goal's arguments are.

*e-goal-pred-in-direct-line* returns *true* if the argument is the predicate of the current goal or one of its ancestors and *false* otherwise. (We look at the predicate because we usually don't care what the goal's arguments are.)

### 5.3 Random choices

*e-random* takes one argument (*n*) and succeeds (without bindings) *n%* of the time. If you have multiple operators which satisfy the same goal, make sure that at least one will succeed, i.e., if operators have otherwise identical preconditions, don't put *e-random* on the one which will be executed last (i.e., the first one in the file).

*e-random-pick* takes two arguments, a list and a variable, and unifies a random entry in the list with the variable. It fails if the first argument is not a list, the second argument is not a variable, or the list is empty.

## 5.4 Forms of selection

*e-unify* unifies two patterns. If they can be unified, it returns a binding set containing the binding list. If no bindings are required, it returns *true*. If unification fails, it returns *false*. (This is the same protocol as the APE internal function *show*.)

Examples:

```
(e-unify ?x 3)           --> (((x 3)))
(e-unify ?x ?y)         --> (((x ?y)))
(e-unify '(?x ?y) '(3 ?z)) --> (((x 3) (y ?z)))
(e-unify 3 3)           --> ((nil))
(e-unify 3 4)           --? nil
```

*e-equals* checks to see whether two items have the same value. This is intended to be used for a variable and a constant or perhaps two variables, but works for any two items. It unifies the two items and returns a *true* if they unify. If they don't, it returns *false*. It does not return the actual binding list (use *e-unify* for that).

## 5.5 List processing

*e-atom* takes one argument and returns *true* if the argument is an atom and *false* otherwise.

*e-cons* takes three arguments, an atom and two lists. It assumes the atom and the first list are instantiated. If the second list is also instantiated, it returns *true* if it is the result of *cons*-ing the atom onto the first list and *false* if it isn't. If the second list is a variable, it binds the variable to the result of *cons*-ing the atom onto the first list.

*e-append* takes three arguments, all lists. It assumes the first two are instantiated. If the third is also instantiated, it returns *true* if the third is the result of appending the first two and *false* if it isn't. If the third is a variable, it binds the variable to the result of appending the first two.

*e-head* takes two arguments, a list and an item. If both are instantiated, it returns *true* if the item is the first element of the list and *false* if it isn't. If the second argument is a variable, it binds the variable to the head of the list.

*e-length* takes two arguments, a list and a length. If both are instantiated, it returns *true* if the length of the first argument equals the second and *false* if it doesn't. If the second argument is a variable, it binds the variable to the length of the list.

*e-listify* takes two arguments, an atom and a list. If both are instantiated, it returns *true* if the second argument is a list containing the first and *false* if it isn't. If the second argument is a variable, it binds the variable to a list containing the first argument.

*e-listp* takes one argument and returns *true* if the argument is a list and *false* otherwise.

*e-member* checks to see whether the first item is a member of the second, then returns *true* or *false*. It does not return the tail the way *member* does.

*e-reverse* takes two arguments, both lists. If both are instantiated, it returns *true* if the second argument is the reverse of the first and *false* if it isn't. If the second argument is a variable, it binds the variable to the reverse of the list.

*e-tail* takes two arguments, both lists. If both are instantiated, it returns *true* if the second argument is the tail of the first and *false* if it isn't. If the second argument is a variable, it binds the variable to the tail of the list. If the first argument is *nil*, *e-tail* fails. This property is useful for ending a recursive traversal of a list.

## 5.6 Arithmetic

*e-sum* takes three atoms as arguments, of which either two or three are instantiated. If all three are instantiated, it returns *true* if the third is the sum of the first two and *false* if it isn't. If one is a variable, it binds the variable to the value required to make  $v1 + v2 = v3$  true.

*e-greater* takes two instantiated items as arguments. If the first is greater than the second, it returns *true*. Otherwise it returns *false*.

## 5.7 Passthrough to underlying Lisp system

*e-execute* takes a Lisp form as input and returns a binding list containing its value. The result is protected inside an extra pair of parentheses to prevent the result from ever being *nil*. This function is useful for calling Lisp functions inside APE.

## 6. References

Dijkstra, E. W. 1972. Notes on Structured Programming. In Dahl, O.-J., Dijkstra, E. W. and Hoare, C. A. R., *Structured Programming*. London: Academic Press.

Wilkins, D. 1988. *Practical Planning: Extending the Classical AI Planning Paradigm*. San Mateo, CA: Morgan Kaufmann.