1 Introduction

The shell, in file '`shellOAVbc.pro`', implements a simple backward-chaining object-attribute-value expert system shell. Because the shell is intended to be simple rather than complete there are many obvious ways the shell could be extended; some of these will appear as homework or test questions. This document describes how rules for the shell are written.

2 Data Types

2.1 Overview

All data is stored as object-attribute-value (OAV) triples. OAVs can either be *static* or *instantiated*, created dynamically. The identifiers (symbol for the "O" part) for static objects, since they are known before runtime, can appear in rules. For the same reason attribute values for static objects can be specified along with with rules. This is done using four-element structures of the form `data(`*Object*,*Attribute*,*Value*,`init)`. For example,

```
data(car,wheels,4,init).
data(car,fuel,gas,init).
```

might appear in a file with rules about cars. If the identifier for an object is omitted the *global* object is assumed.

2.2 Attribute Properties

Object attributes are associated with *properties*. In the shell attributes can have property *askable*, *announcable*, and a validation. If an attribute is askable the user can be asked to provide its value; if it has a validation it is used to check the answer. (See Section 5.2 [Miscellaneous], page 9 function `askAndCheck`.)

An attribute `A` of static object or class `O` is given the announcable property by placing `A:.O` in a list in the single component of Prolog fact `aGoalList`, usually placed in the rule file. For example, with

```
aGoalList([car:.repaired,tigers:.won]).
```

in the rules file, when `car:.repaired` or `tigers:.won` is assigned a value a message is printed for the user. Properties defined for a class apply to all instances of the class.

2.3 Instantiated Objects

An *instantiated object* is created at run time from a class. A class is defined simply by specifying an askable property for one of its attributes. Objects are instantiated from a class by using the `new` function. See Section 5.1 [Instance Functions], page 8.

3 Rules

Rules have three elements, an *id*, *antecedent*, and *consequent*:

id:if *antecedent* then *consequent*.

The rule id is for users' convenience. The antecedent is an expression that evaluates to a truth value. The consequent either specifies an OAV or a command, something like a function declaration. In the former case an assignment occurs if the antecedent is true, in the later case no special action is taken even if the antecedent is true.

The value to be associated with the consequent can be explicitly shown by using the assignment operator, '`:=`'. The left operand is the object and attribute to be assigned, the right operand is a variable. The value is the binding of the variable after the antecedent is evaluated. (See examples below.)

If the assignment operator is absent a truth value is associated with the consequent. The truth value is '`True`' if the antecedent evaluates to '`true`'. If the consequent is an OA (not a command) then a '`False`' is assigned if a value cannot be determined in any other way. That is, if all antecedents are false and if the attribute does not have the askable property.

The following chapters describe operators and functions used to form the expressions; consequents will be described here.

Consider a consequent containing an attribute of the global object and an attribute of a static object:

```
r2:if a and b then c.
r2b:if a and b then d:.e.
```

In rule r2 '`c`' is assigned value '`True`' if '`a`' and '`b`' are true; in rule r2b attribute `e` of object `d` is assigned a truth value. (Note that `a`, `b`, and `c` are attributes of the global object and so are equivalent to `global:.a`, `global:.b`, and `global:.c`.) If the antecedents of r2 and r2b are false, and if there is no other way to determine values for `c` and `d:.e`, then they will be assigned the value '`False`'.

A consequent of the form `Obj:.Attr:=Val` specifies that value `Val` is to be assigned if the antecedent is true. Both `Obj` and `Attr` must be bound before the antecedent is evaluated, `Val` could be bound as a result of the antecedent evaluation. The value assigned is remembered and so the rule would not be used again for the same object and attribute. For example,

```
r3:if coffeeBeans:.costPerKilo=C and purchase:.amount=A
          and Price is C*A then purchase:.price:=Price.
r4:if purchase:.price=Price and purchase:.price=P2 then twice.
```

in rule r3 `Price` is computed in the antecedent and assigned in the consequent. Rule r4 might cause rule r3 to execute once, even though `purchase:.price` appears twice in the antecedent; the second time the value is retrieved from the `purchase` object.

Other valid Prolog expressions can be placed in the consequent of rules, these are called *commands* or *functions* and are treated something like function calls. The antecedent of the rule is evaluated, perhaps binding some parts of the Prolog expression to values. Multiple rules can have the same commands in their consequent; if the antecedent for one such rule evaluates to false the next one will be tried; an error results if the antecedents of all such rules are false. Unlike value assignment, each time an expression appearing in a rule consequent is encountered the corresponding rule's antecedent will be evaluated. For example,

```
r5:if coffeeBeans:.costPerKilo=C and purchase:.amount=A
          and Price is C*A then compPrice(Price)
r6:if compPrice(P) and compPrice(P2) then twice.
```

here when r6's antecedent is evaluated, the antecedent of r5 is evaluated twice. While the shell will accept almost any Prolog expression in the consequent, those of interest are structures with a few components.

4 Operators

Logical, unification, and arithmetic computation and comparison operations can be used in expressions. Some work identically to their Prolog equivalents (in fact, they're implemented by passing expressions to the Prolog interpretor).

and: *E1* `and` *E2* Operator

Operator '`and`' (priority 750, precedence '`xfy`') returns true if *E1* and *E2* are true.

For example,

```
andExampleRule:if a and b then c.
```

the expression in the antecedent evaluates to '`True`' if `a` and `b` evaluate to '`True`'.

or: *E1* `or` *E2* Operator

Operator '`or`' (priority 800, precedence '`xfy`') returns true if *E1* or *E2* are true.

For example,

```
orExampleRule:if a or b and c then d.
```

the expression in the antecedent evaluates to '`True`' if `a` evaluates to '`True`' or `b` and `c` evaluate to '`True`' (note higher precedence of `and`).

not: `not` *E* Operator

Operator '`not`' (priority 700, precedence '`fy`') return true if *E* is false.

For example,

```
notExampleRule:if not a or b and c then d.
```

the expression in the antecedent evaluates to '`True`' if `a` evaluates to '`False`' or `b` and `c` evaluate to '`True`' (note precedence of `not` is higher than '`or`').

<: *E1* `<` *E2* Operator

=<: *E1* `=<` *E2* Operator

>: *E1* `>` *E2* Operator

>=: *E1* `>=` *E2* Operator

Each of these operators have priority 700 and precedence '`xfx`'. They perform the comparison on their two operands, which must be numeric expressions, the return value is the result of the comparison.

For example,

```
compExamp:if truck:.numWheels=L and L>18 then truck:.huge:=true.
```

antecedent is true if a value for attribute '`numWheels`' can be found and that value is greater than 18.

is: V `is` E Operator

Operator '`is`' evaluates E as a numeric expression and binds the result to V.

For example,

```
isExample:if coffeeBeans:.costPerKilo=C and purchase:.amount=A
          and Price is C*A then purchase:.price:=Price.
```

the antecedent here computes the price of coffee based on an amount and a price per kilogram.

5 Functions (Commands)

Functions are used in expressions to compute values or perform actions. Most evaluate to '`True`' or '`False`' while returning values by binding variables. The instance functions are used for instantiation and for operating on instantiated objects. The miscellaneous function is used for asking the user a question; the low level functions might be used for extending or circumventing features of the shell.

5.1 Instance-Related

forAll(*Object*'`:.isa=`'*Class, Inclusion, Condition*) Function
forEach(*Object*'`:.isa=`'*Class, Inclusion, Condition*) Function
forSome(*Object*'`:.isa=`'*Class, Inclusion, Condition*) Function

Generates a set of instances of class *Class* for which expression *Inclusion*, which includes *Object*, is '`true`'. Next, evaluates *Condition* for each instance in the set. Function '`forAll`' returns '`True`' if all evaluate to '`True`' or if the set is empty; function '`forEach`' always returns true; and function '`forSome`' returns true if at least one evaluates to '`True`'.

For example,

```
callVetRule:if forSome( Animal:.isa=zooAnimal, Animal:.alive=true,
               Animal:.sick=true ) then vetNeeded.
```

the antecedent of this rule is true if there are any live zoo animals that are sick. Using the terminology in the definition, the set of instances for which *Inclusion* (`Animal:.alive=true`) is '`True`' is the set of all live zoo animals; if *Condition* (`Animal:.sick=true`) is true for at least one, unfortunate, animal '`forSome`' returns true. Function '`forAll`' would return true if there were no animals or they were all dead, which is not what we want. Function '`forEach`' might be used to force an expression to be evaluated for each object.

new(*Class, Instance*) Function

Bind *Instance* to a new instance of class *Class*. Returns '`True`' if instance is created; '`False`', otherwise.

For example,

```
newPerson:if new(people,Person) and Person:.name=_ and Person:.age=_
          then newPerson(Person).
```

this rule creates a new instance of the class people and forces the system to determine the person's name and age, which presumably have the askable property. Note that the consequent of the rule contains a command, not a proposition or object attribute.

thereExist(*Object*`:.isa=`*Class*, *SuchThat*) Function

Binds *Object* with an object of class *Class* for which expression *SuchThat* is true. Normally *SuchThat* contains *Object*; see examples below.

For example,

```
wereSaved:if thereExist( P:.isa=person,
                         P:.canFightBear and P:.brave )
          then unsuspectingHero:=P.
```

if there is an instance of the class person for which attributes '`canFightBear`' and '`brave`' are true then `unsuspectingHero` is assigned. Consider two inadequate alternatives :

```
wereSaved1:if thereExist( P:.isa=person, P:.canFightBear )
              and P:.brave
           then unsuspectingHero:=P.
wereSaved2:if forSome( P:.isa=person, P:.canFightBear, P:.brave )
              then unsuspectingHero:=P.
```

Rule '`wereSaved1`' won't always work because the first `P` found may not be brave, and so the antecedent would fail. (Unlike Prolog the shell doesn't backtracking.) Rule '`wereSaved2`' won't work because unlike '`thereExist`', '`forSome`' will not bind `P` to an instance outside of '`forSome`' (`P` will remain unbound).

5.2 Miscellaneous

askAndCheck(*QuestionText, Subject, Valid, Answer*) Function

Binds *Answer* with response of user to question text *QuestionText* subject to validation *Valid*. An '*' (asterisk) in *QuestionText* (if any) is replaced by *Subject*. Parameter *Valid* can be `isEnum(`*List*`)`, `isNum(`*Limits*`)`, `yesNo`, or `isString`. When called with `yesNo`, only a yes or no answer is accepted. When called with `isString` any string is accepted. When called with `isEnum` the answer must be a member of the list *List*. When called with `isNum` the answer must be a number in the range specified in *Limits*. *Limits* can be `none`, `range(`*min*,*max*`)`, `min(`*min*`)`, or `max(`*max*`)`. See the code below or code in the file '`ask.pro`' for details.

For example,

```
passRule:if  askAndCheck("Do you want to cross the bridge?",_,yesNo,Ans)
            and
             Ans=yes
            and
             askAndCheck("What is your name?",_,isString,Name)
            and
             askAndCheck("*, what is your quest?",Name,
              isEnum([grail,otherSide,stroll]),Quest)
            and
             askAndCheck("What is the airspeed of a swallow",_,
              isNum(range(0,100)),Speed)
            and
             okay(Name,Quest,Speed)
           then
            canCrossBridge.
```

the antecedent of this rule can result in four questions being asked. The first gets a yes-or-no answer, the second will accept any string (for the name), the third question will accept three possible answers (asked as a menu of three choices), and the fourth will accept a number in the range of 0 to 100. Note that the third question includes the user's name. Command `okay` might be in the consequent of a rule that determines if person can cross.

5.3 Low-Level Functions

call(*PrologCode*) Function

Have Prolog evaluate *PrologCode* and return '`True`' if it succeeds; '`False`', otherwise.

For example,

```
r1:if Transport:.type=T and Transport:.passengers=P
      and call( retract(transList(L)) )
      and call( asserta(transList( [ tp(T,P) | L ] )) )
   then Transport:.processed.
```

this rule uses '`call`' to add information about transport objects to a list within a Prolog fact.

conclude(*Object:.Attribute, Value, Source*) Function

Assign *Value* to *Attribute* of *Object* and remember that *Value* was determined by *Source*. Announce value assigned if *Attribute* of *Object* or *Objects*'s class has announce property. This function is automatically called when a value is assigned in the consequent of a rule.

newName(*Name, NextName*) Function

'`newName`' binds *NextName* to a new identifier consisting of the characters in atom *Name* with a number appended. The number is the number of times '`newName`' was called with *Name*. (That is, 1 the first time *Name* is used, 2 the second time, and so on.) This function is used to make identifiers for new instances.

Index

Table of Contents