A Brief Introduction to Epikit

1 Introduction

EPIKIT is a library of Common Lisp subroutines intended for use in the construction of programs that operate on databases encoded in Knowledge Interchange Format (KIF).

This document is a brief introduction to some of the key features of EPIKIT. The presentation here assumes that the reader is familar with KIF, with automated inference techniques, and with partial programs. For more information on KIF, the reader is referred to the *Knowledge Interchange Format Reference Manual.* For more information on automated inference and partial programs, the reader is referred to any good AI text. The implementation of these capabilities in EPIKIT is modelled on the treatment in *Logical Foundations of Artificial Intelligence.*

2 Database Manipulation

First and foremost, EPIKIT provides capabilities for managing databases of KIF sentences. In this section, we introduce the basic subroutines for creating, examining, and destroying databases. Included is an introduction the product's heterarchical theory mechanism.

The `save` command is used to add facts to the current "theory". Here, we add the facts that Art is the parent of Bob, Bea, and Bess.

```
User: (save '(parent art bob))
Lisp: (PARENT ART BOB)

User: (save '(parent art bea))
Lisp: (PARENT ART BEA)

User: (save '(parent art bess))
Lisp: (PARENT ART BESS)
```

Once we have some facts in our database, we can examine its contents. The **knownp** command takes a sentence as argument. If the sentence is ground, then **knownp** succeeds if and only if the database contains that sentence. If the sentence includes individual or sequence variables, **knownp** succeeds if and only if the database contains a sentence that matches the argument for some binding of the constituent variables.

```
User: (knownp '(parent art bob))
Lisp: T

User: (knownp '(parent $x bob))
Lisp: T

User: (knownp '(parent @l))
Lisp: T
```

The **knownx** command takes an expression and a sentence as argument. If the database contains a matching sentence, **knownx** plugs the variable bindings (if any) into the specified expression and returns the answer.

There is no restriction on the position of the variables in the query. Note that there are three possible answers for the second query. The `knownx` subroutine always returns the first answer it finds.

```
User: (knownx '$x '(parent $x bob))
Lisp: ART

User: (knownx '$y '(parent art $y))
Lisp: BOB

User: (knownx '(relatives @l) '(parent @l))
Lisp: (RELATIVES ART BOB)
```

The `knowns` subroutine is similar to `knownx` except that it returns a list of all possible answers. The order of answers on the list is the order in which they are found in the database.

```
User: (knowns '$y '(parent art $y))
Lisp: (BOB BEA BESS)

User: (knowns '($x $y) '(parent $x $y))
Lisp: ((ART BOB) (ART BEA) (ART BESS))
```

The `drop` subroutine is used to delete sentences from the database. If `drop` succeeds in finding a matching sentence to remove, it returns the sentence as value.

```
User: (drop '(parent art bess))
Lisp: (PARENT ART BESS)

User: (knowns '$y '(parent art $y))
Lisp: (BOB BEA)
```

An EPIKIT database is broken into chunks called *theories*. Every theory has a symbol as its *name* and has a sequence of sentences as its *contents*. By simply mentioning the name of a theory, it is possible to make the contents available or unavailable for access, addition, and deletion.

At each point in time, there is exactly one *current* theory, and all database accesses, additions, and deletions take place with respect to that theory. In EPIKIT, the current theory is determined by the value of the variable `*theory*`. Its initial value is `global`. If we look at the contents of `global`, we see the sentences we added earlier.

```
User: *theory*
Lisp: GLOBAL

User: (contents 'global)
Lisp: ((PARENT ART BOB) (PARENT ART BEA))
```

By changing the value of `*theory*`, we can add sentences to other theories. For example, here we change the value of `*theory*` to `mytheory` and add some sentences. Notice that, if we try to look up facts after these changes, only the new sentences are available.

```
User: (setq *theory* 'mytheory)
Lisp: MYTHEORY

User: (save '(parent art bill))
Lisp: (PARENT ART BILL)

User: (save '(parent art betty))
Lisp: (PARENT ART BETTY)

User: (knowns '$y '(parent art $y))
Lisp: (BILL BETTY)
```

We can access or modify our old theory by switching the value of `*theory*` back again or by passing an additional argument to any database command. In the following example, we access our old data by specifying `global` in our call to `knowns`.

```
User: (knowns '$y '(parent art $y) 'global)
Lisp: (BOB BEA)
```

Another way of creating a theory is to use the `deftheory` command. This subroutine takes a theory name and a list of sentences as arguments and arranges that the theory contains exactly those sentences specified and no more. Here, we create a theory called `another` with two more facts about Art.

```
User: (deftheory another
User:   (parent art ben)
User:   (parent art barbara))
Lisp: ANOTHER
```

Often in working with theories, it is useful to include the facts from one theory inside of another theory. We can do this by using the `save` command to add them in the other theory as well, but this is wasteful. An alternative is to call the `includes` routine on two theories, thereby telling the database routines that the first theory implicitly includes the second. Here, we say that `mytheory` includes `global` and thereby make all of the facts in `global` available whenever a subroutine accesses `mytheory`. The `includees` subroutine returns a list of all theories included in the theory specified as argument. The variable `*includers*` always contains a list of theories with included theories.

```
User: (includes 'mytheory 'global)
Lisp: DONE

User: (includes 'mytheory 'another)
Lisp: DONE

User: (knowns '$y '(parent art $y))
Lisp: (BILL BETTY BOB BEA BEN BARBARA)

User: (includees 'mytheory)
Lisp: (GLOBAL ANOTHER)

User: *includers*
Lisp: (MYTHEORY)
```

The variable `*theories*` contains a list of theories containing one or more sentences. `empty` removes all sentences from a given theory. `reset` resets the state of the database and all variables in EPIKIT to their initial values.

```
User: *theories*
Lisp: (ANOTHER MYTHEORY GLOBAL)

User: (empty 'global)
Lisp: DONE

User: *theories*
Lisp: (ANOTHER MYTHEORY)

User: (reset)
Lisp: DONE
```

3 Inference

In this section, we present some of the reasoning programs available in EPIKIT. We start off with a look at the `prolog` series and then introduce a number of variations.

The database manipulation routines introduced above are unable to answer even the simplest questions of arithmetic or equality (unless those facts are explicitly stored in the database). By contrast, the `prolog` subroutines have knowledge of many basic concepts. The `prologp` subroutine takes arguments and returns values in a manner analogous to that of `knownp`; `prologx` is analogous to `knownx`.

```
User: (knownp '(> 3 2))
Lisp: NIL

User: (knownp '(= a a))
Lisp: NIL

User: (prologp '(> 3 2))
Lisp: T

User: (prologp '(= a a))
Lisp: T

User: (prologx '$x '(= $x a))
Lisp: A
```

The special strength of the `prolog` subroutines is their ability to do inference with logical information encoded as implications. Here, we enter a definition for the `grandparent` relation, and we enter some facts about Art's family. Although, according to our definitions, Art is the grandparent of Cal, `knownp` answers `nil`. This is the correct answer for `knownp` – after all, the fact is not in the database. By contrast, `prologp` is able to prove the fact. The `prologx` subroutine is able to find a grandparent of Cal and a grandchild of Art. And the `prologs` subroutine is able to find all of the grandchildren of Art.

```
User: (save '(<= (grandparent $x $z) (parent $x $y) (parent $y $z)))
Lisp: (<= (GRANDPARENT $X $Z) (PARENT $X $Y) (PARENT $Y $Z))
```

```
User: (save '(parent art bob))
Lisp: (PARENT ART BOB)

User: (save '(parent bob cal))
Lisp: (PARENT BOB CAL)

User: (save '(parent bob coe))
Lisp: (PARENT BOB COE)

User: (knownp '(grandparent art cal))
Lisp: NIL

User: (prologp '(grandparent art cal))
Lisp: T

User: (prologx '$x '(grandparent $x cal))
Lisp: ART

User: (prologx '$y '(grandparent art $y))
Lisp: CAL

User: (prologs '$y '(grandparent art $y))
Lisp: (CAL COE)
```

The `prolog` subroutines introduced so far use an inference technique that is essentially identical to the depth-first backward chaining technique used in the PROLOG interpreter. One of EPIKIT's extensions to PROLOG is forward chaining. By writing an implication with a forward arrow, we prescribe that the implication is to be used in the forward direction only.

As an example, consider the following interaction. The forward implication asserts that, if a person is in a particular family, then all of his children are in that family as well. By writing it as a forward implication, we are saying that the implication should be triggered whenever we get information about a person's family. The subroutine `prolog` is used to add facts to the database and conduct all such forward chaining. If we discover that Art is a Garfunkel, then we immediately conclude that Bob and Cal and Coe are Garfunkels as well and these facts are stored explicitly in the database.

```
User: (save '(==> (family $x $z) (parent $x $y) (family $y $z)))
Lisp: (==> (FAMILY $X $Z) (PARENT $X $Y) (FAMILY $Y $Z))

User: (prolog '(family art garfunkel))
Lisp: DONE

User: (knowns '$x '(family $x garfunkel))
Lisp: (ART BOB CAL COE)
```

Like PROLOG, the `prolog` subroutines offer nonmonotonic inference based on failure. However, this capability is not attached to the negation operator, as in PROLOG. If we ask whether it is the case that Art is *not* the grandparent of Chris, we get `nil`, meaning that `prologp` is unable to determine the truth of this sentence. In fact, Art *might* be the grandparent of Chris, and we just do not know it. To get the effect of PROLOG's failure mechanism, we use the operator `provable`. If we do not know that Art is the grandparent of Chris, then it is certainly *not* provable.

```
User: (prologp '(not (grandparent art chris)))
Lisp: NIL

User: (prologp '(not (provable (grandparent art chris))))
Lisp: T
```

There are two other nonmonotonic operators known to `prolog`. The `bagof` operator takes a term and a sentence as arguments. It is equivalent to a list of all instances of the term for which the sentence is provable. The `oneof` operator also takes a term and a sentence as arguments. It is equivalent to the first instance of the term for which the sentence is provable.

```
User: (prologx '$l '(= $l (bagof $z (grandparent art $z))))
Lisp: (LIST CAL COE)

User: (prologx '$x '(= $x (oneof $z (grandparent art $z))))
Lisp: CAL
```

Procedural attachments are often nonmonotonic in that their results can change depending on the time at which they are processed. In processing a term involving the function `eval`, the `prolog` subroutines evaluate the argument of the term and use the result of this evaluation as the value of the term. We can use this capability, for example, to specify a particular inference procedure to use in evaluating a sentence.

```
User: (prologx '$l '(= $l (eval '(knowns '$z '(grandparent art $z)))))
Lisp: 'NIL

User: (prologx '$l '(= $l (eval '(prologs'$z '(grandparent art $z)))))
Lisp: '(CAL COE)
```

So far, we have concentrated exclusively on relations. Much, if not most, of our conceptualization of the world naturally takes the form of functions. For functional information, we use a notation much closer to that of Lisp. In the first sentence below, we define the children of a person to be the bag of all individuals of whom the person is a parent. The length of a list is 0 if the list is empty; otherwise, it is one more that the length of the remainder of the list. (Note that, here and hereafter, we leave off recording Lisp's responses to `save` commands.)

```
User: (save '(<- (children $x) (bagof $y (parent $x $y))))

User: (save '(<- (length (list)) 0))

User: (save '(<- (length (list $x @l)) (1+ (length (list @l)))))
```

The `prolog` series of subroutines also includes a subroutine, called `proval`, capable of evaluating terms. (It is also used in evaluting the arguments to equations within calls to `prolog`, `prologp`, `prologx`, and `prologs`)

```
User: (proval '(length (children bob)))
Lisp: 2
```

Sometimes, it is desirable to intermix functional and relational concepts. In the following sentence, we define a person to be prodigious if he is a parent with more than one child.

```
User: (save '(<= (prodigious $x)
User:            (parent $x $y)
User:            (> (length (children $x)) 1)))
```

Unfortunately, we now encounter a weakness of PROLOG and, consequently, the `prolog` series of reasoning programs. They are unable to evaluate terms inside of implications, in this case the `length` expression.

```
User: (prologp '(prodigious bob))
Lisp: NIL
```

It is possible to get a term t evaluated by adding to the implication an equational condition of the form `(=` t `$n)` and then rewriting the inequality as `(> $n 1)`. However, when implications get large, this can be cumbersome. To simplify matters, EPIKIT includes several additional series of subroutines for handling equalities without requiring this sort of distortion. The `paralog` series behaves just like the `prolog` series except that it evaluates terms as it goes.

```
User: (paralogp  '(prodigious bob))
Lisp: T

User: (paralogx '$x '(prodigious $x))
Lisp: BOB
```

The `paralogp` subroutine is similar to the Lisp interpreter in that it proceses terms in bottom up fashion. However, it is unlike Lisp in that it can handle variables in the terms being evaluated in a manner similar to that used by `prolog` in processing sentences with variables. In order to do this, it sometimes needs to invert function definitions, i.e. to compute what inputs would lead to a given output, as in the following example. Note that the components of the list are left unspecified.

```
User: (paralogx '$l '(= (length $l) 3))
Lisp: (LIST $G1 $G2 $G3)
```

In addition to the `paralog` series, there are several other series, each capable of handling equality on a different calss of problems. The `basic` series is used whenever the query and database are both ground. The `demodulog` series is used when the database contains variables but the query and all subgoals are ground. The `basic` series is faster than the `demodulog` series, and the `demodulog` series is faster than the `prolog` and `paralog` series.

While all of these procedures are sound and complete on various subsets of predicate calculus, none is complete on the entire language. (The `paralog series` comes very close.) However, there are some additional subroutines that provide this assurance. Given any database and any query in first order predicate calculus with equality, the `epilog` series will answer `t` if and only if the query is logically implied by the database.

```
User: (epilogidp '(or p (not p)))
Lisp: T
```

The inference technique used in the `epilog` series is an implementation of model elimination and paramodulation applied in iterative deepening fashion.

4 Partial Programs

One of the primary uses of declarative databases is the control of agents, i.e. entities that can observe and act as well as reason. Robots are good examples of agents. They observe their environments via touch sensors, video cameras, infrared detectors, etc.; and they act on their environments by moving their arms, their grippers, their wheels, etc. Expert consultation systems can also be viewed as agents. They print out questions and results, and they read the answers entered by the user. Even individual Lisp programs can profitably be thought of as agents. They sense their environments (e.g. by accessing global variables), and they are capable of having side effects on those environments (e.g. by setting the values of global variables).

An agent's observation of its environment at each point in time is one of a fixed (but possibly very large) set of *percepts* (determined by its sensory hardware). Its effect on the environment at each point in time is one of a predetermined set of *actions* (determined by its effectory hardware). In the context of Lisp, actions are individual Lisp subroutine calls, and percepts are the results of executing those subroutines. For example, the act of applying Lisp's *read* subroutine is an action, and the resulting value is the subsequent percept.

For a machine with a set A of actions, we define a procedure to be a function from the natural numbers into A, which maps each time step into the action to be performed on that step. A *generalized procedure* is a function that maps each time step into an action *or* another procedure (which is run to completion before considering the next step of the outer procedure).

A *KIF program* is a description of a generalized procedure in KIF. In writing a KIF program, we select a term (usually a constant) to denote the procedure being defined. We then write axioms characterizing the function constituting the procedure.

EPIKIT provides an interpreter for KIF programs, called `interpret`. It takes a term and a theory as arguments and executes the procedure denoted by the specified term and described in the specified theory. The operation of the interpreter is a simple "observe-think-act" loop. On each cycle of operation, it observes the environment to get a percept x and draws conclusions by forward chaining from a sentence of the form `(= (percept` p n`)` x`)`, where p is the name of the procedure and n is the number of the cycle. It then computes an action to be executed by backward chaining to find a value for a term of the form `(action` p n`)`, where p is the name of the procedure and n is the number of the cycle. If the action is a `return` action, the interpreter halts and returns the argument of the `return` action. If no action is found, the interpreter halts unsuccessfully. Otherwise, the action is executed, and the cycle repeats.

As an illustration of this partial program interpreter in action, consider the problem of controlling the pressure in a reservoir. Let us assume that the reservoir has an inlet that adds material (and thereby tends to drive the pressure up), a drain that removes material (and thereby tends to drive the pressure down), and a customer that also removes material (and thereby tends to drive the pressure down). In order to control the pressure, we have a number of actions at our disposal. We can temporarily close the inlet; we can temporarily open the drain; or we can do nothing. Each action is implemented as a Lisp subroutine, and each returns the current pressure in the reservoir as value.

We start off by writing a sentence to interpret our machine's percepts. If the value on a step exceeds 100 psi, then we know that a high pressure situation exists.

```
User: (save '(==> (= (percept program-1 $n) $z)
                  (> $z 100)
                  (high-pressure $n)))
```

This fact could also have been written as a backward implication. We have written it here as a forward implication to illustrate that the interpreter uses a mixture of forward chaining from observed data (based on facts like this one) and backward chaining from the goal of finding an action (as illustrated below).

In order to characterize our program `program-1`, we write a single general axiom. The action on the n-th step of operation is any action that is `doable` and cannot be shown to be `bad`. The concepts `doable` and `bad` are axiomatized below. (The interpreter knows nothing about these concepts except for the axioms provided by the KIF programmer.)

```
User: (save '(<- (action program-1 $n)
                 (oneof $a (and (doable $a $n)
                                (not (provable (bad $a $n)))))))
```

As mentioned in the problem description, there are three possible actions. By asserting the following sentences, we are stating that these actions are doable on every step of operation.

```
User: (save '(doable (call 'noop) $n))

User: (save '(doable (call 'shut-inlet) $n))

User: (save '(doable (call 'open-drain) $n))
```

In some reasoning systems, the mention of an action automatically leads to the execution of that action, whenever that occurrrence is processed by the reasoning system. That is not the case here. The term `(call 'shut-inlet)` *refers* to the action of shutting the inlet. Reasoning with the term does not automatically cause the action to take place. Only `interpret` executes actions; the reasoning programs do not.

Although all three of our actions are doable, some are better than others in certain situations. For example, it is a bad idea to do nothing if the reservoir exhibits high pressure.

```
User: (save '(<= (bad (call 'noop) $n)
                 (high-pressure $n)))
```

Another way for an action to be bad is if there is a doable action that is preferred. Closing the inlet is better than opening the drain, because it does not waste material.

```
User: (save '(<= (bad $b $n)
                 (doable $a $n)
                 (better $a $b)))

User: (save '(better (call 'shut-inlet) (call 'open-drain)))
```

Now, we can run our program. Since we do not have a real reservoir at our disposal, we can simulate one by defining our actions so that their sole effects are to make adjustments on a fictional pressure variable. In this case, we use the atom `*pressure*` and initialize the pressure to be 95 pounds per square inch. By executing the `noop` action, *we* are not changing the state of affairs in the reservoir. However, the pressure still rises due to the open inlet. By executing the `shut-inlet` action, we stop the pressure rise, and the customer's demand on the reservoir causes the pressure to fall 30 pounds per square inch. Similarly, by executing the `open-drain` action, we stop the pressure rise, and the customer's demand causes the pressure to fall 30 pounds per square inch.

```
User: (defvar *pressure* 95)
Lisp: *PRESSURE*
```

```
User: (defun noop () (setq *pressure* (+ *pressure* 10)))
Lisp: NOOP

User: (defun shut-inlet () (setq *pressure* (- *pressure* 30)))
Lisp: SHUT-INLET

User: (defun open-drain () (setq *pressure* (- *pressure* 30)))
Lisp: OPEN-DRAIN
```

With these definitions in place, we can test out our partial program. First, we ask the system to print out a message indicating the percept and action on every step of operation.

```
User: (trace-percept '$x)
Lisp: DONE

User: (trace-action '$x)
Lisp: DONE
```

Now, we call `interpret` on our program and watch the results. The initial percept is 0. (Remember that a percept in Lisp is the value returned by the preceding action. Since there is no preceding action on the first step, this percept is simply a default value.) The first action in this case is `noop`. This results in a pressure rise to 105 psi, the percept for the second step. In this case, `noop` is a bad action, due to the high pressure; and, following its program, the system decides to temporarily close the inlet. This results in a pressure drop on the third step, leading to a sequence of `noop` actions. And so forth.

```
User: (interpret 'program-1)

Percept: 0
Action:  (CALL 'NOOP)

Percept: 105
Action:  (CALL 'SHUT-INLET)

Percept: 75
Action:  (CALL 'NOOP)

Percept: 85
Action:  (CALL 'NOOP)

Percept: 95
Action:  (CALL 'NOOP)

Percept: 105
Action:  (CALL 'SHUT-INLET)

*** Computation aborted ***

User: (reset)
Lisp: DONE
```

If the system does not behave as desired, it is possible to tune its behavior by adding additional facts to the theory describing our procedure. For example, we may get information about additional actions that are doable; we may get more information about which actions are bad in and of themselves; and we may get more information about which actions are better than others.

While this example is very simple, partial programming methodology can be used in extremely complicated applications as well. It allows the smooth blending of seemingly disparate programming techniques – such as Lisp programming, object-oriented programming, and production systems. The next section illustrates the use of partial programming on a more complicated example.

5 Metaknowledge

One of the distinctive features of KIF is that it allows us to express knowledge about knowledge. In order to encode such *metaknowledge*, we use a conceptualization in which expressions in the language are treated as objects in the universe of discourse and in which there are functions and relations appropriate to these objects. In KIF, atoms are treated as primitive objects (i.e. having no subparts). Complex expressions (i.e. non-atoms) are treated as lists of subexpressions (either atoms or other complex expressions). In particular, every complex expression is viewed as a list of its immediate subexpressions.

In order to assert properties of expressions in the language, we need a way of referring to those expressions. There are two ways of doing this in KIF. One way is to use the `quote` operator in front of an expression. For example, to refer to the symbol `john`, we use the term `'john` or, equivalently, `(quote john)`. To refer to the expression `(p a b)`, we use the term `'(p a b)` or, equivalently, `(quote (p a b))`.

With a way of referring to expressions, we can assert their properties. For example, the following sentence ascribes to the individual named `john` the belief that the moon is made of a particular kind of blue cheese.

```
User: (save '(bel john '(material moon stilton)))
Lisp: (BEL JOHN '(MATERIAL MOON STILTON))
```

Since expressions are first-order objects, we can quantify over them, thereby asserting properties of whole classes of sentences. For example, we could say that Mary believes everything that John believes. This fact together with the preceding fact allows us to conclude that Mary also believes the moon to be made of blue cheese.

```
User: (save '(<= (bel mary $p) (bel john $p)))
Lisp: (<= (BEL MARY $P) (BEL JOHN $P))

User: (prologp '(bel mary '(material moon stilton)))
Lisp: T
```

The second way of referring to expressions is KIF is to use the `list` function. We can denote a complex expression like `(p a b)` by a term of the form `(list 'p 'a 'b)`, as well as `'(p a b)`. All of EPIKIT's reasoning routines treat the `quote` and `list` forms of denotation as equivalent. If variables are included in the list form, then any attempt to match the two forms will result in the variables being bound to the appropriate subexpressions.

```
User: (prologp '(= '(material moon stilton)
User:              (list 'material 'moon 'stilton)))
Lisp: T
```

```
User: (prologx '$y '(= '(material moon stilton)
User:                  (list 'material 'moon $y)))
Lisp: 'STILTON
```

In order to change levels of denotation, KIF provides the function constant **denotation** (which gives the denotation of the expression supplied as its argument) and **name** (which gives the standard name of the expression supplied as its argument).

```
User: (proval '(denotation 'stilton))
Lisp: STILTON

User: (proval '(name stilton))
Lisp: 'STILTON
```

The advantage of the `list` representation over the `quote` representation is that it allows us to quantify over parts of expressions. For example, let us say that Lisa is more skeptical than Mary. She agrees with John, but only on the composition of the moon. The first sentence below asserts this fact without specifically mentioning `stilton`. Thus, if we were to later discover that John thought the moon to be made of rocks, then Lisa would be constrained to believe this as well. The second sentence relates our assertion about Lisa's beliefs to the world of real objects, not just symbols.

```
User: (save '(<= (bel lisa (list 'material 'moon $y))
User:            (bel john (list 'material 'moon $y))))

User: (save '(<= (moon-material-to-lisa $z)
User:            (bel lisa (list 'material 'moon $y))
User:            (= (denotation $y) $z)))

User: (prologx '$z '(moon-material-to-lisa $z))
Lisp: STILTON
```

Another use of `quote` and `list` is in the formalization of rules of inference. For example, the following sentence formalizes the ordered resolution rule of inference for ground clauses. If the first literals in two clauses are complementary, then it is legal to conclude the clause consisting of the remaining literals of each clause. (Adding unification to this formalization to handle non-ground clauses presents no serious problems.)

```
User: (save '(<= (resolution (list 'or $p @l)
User:                        (list 'or (list 'not $p) @m)
User:                        (list 'or @n))
User:            (= (append (list @l) (list @m)) (list @n))))
```

Since we mention the list function **append** and it is not known to the basic EPIKIT subroutines, we load its definition from the library file. With this information, it is possible for the system to answer metalevel questions about resolution, like the one shown.

```
User: (load-facts "epilibrary:lists.kif")
Lisp: DONE

User: (paralogx '$r '(resolution '(or p q) '(or (not p) (not r)) $r))
Lisp: (LIST 'OR 'Q '(NOT R))
```

Starting with rules of inference formalized in this way, we can build up to entire inference procedures. One application of this ability is to characterize the beliefs of resource-limited agents. It also gives us a flexible way of specifying inference procedures.

As an example of this latter capability, consider the problem of writing a simple resolution theorem-proving system. The system should be capable of finding the contradiction in databases of ground clauses, such as the following.

```
User: (deftheory database
User:   (or p q)
User:   (or (not p) (not r))
User:   (or (not p))
User:   (or (not q)))
Lisp: DONE
```

Our approach is to write the system as a KIF program, in a manner similar to that described in the preceding section. The difference is that the objects being manipulated are databases, and the actions are database additions and deletions. In this case, we will use the single action **save**.

Our first axiom, as before, defines the action of our program on the n-th step to be any action that is doable and not provably bad. The database on the n-th step is found by applying the **contents** function to the target theory.

```
User: (save '(<- (action program-2 $n)
User:            (oneof $a (and (doable $a $n)
User:                           (not (provable (bad $a $n)))))))

User: (save '(<- (database $n) (eval '(contents 'database))))
```

There are two types of doable actions. Our agent can return **true** if the database contains an empty disjunction (equivalent to a contradiction). It can also add a sentence to the theory, provided that the sentence is a conclusion of two sentences already in the theory.

```
User: (save '(<= (doable (return true) $n)
User:            (member '(or) (database $n))))

User: (save '(<= (doable (call 'save $r 'database) $n)
User:            (member $p (database $n))
User:            (member $q (database $n))
User:            (resolution $p $q $r)))
```

Not all doable actions are equally good. We do not want to waste time adding a sentence to the database if it is already there. We also do not want to do an action if there is a better action to do. It is better to return successfully, if that action is doable, then to go on drawing conclusions.

```
User: (save '(<= (bad (call 'save $r 'database) $n)
User:            (member $r (database $n))))

User: (save '(<= (bad $b $n)
User:            (doable $a $n)
```

```
User:           (better $a $b)))

User: (save '(better (return $x) (call 'save $p 'database)))
```

We now have enough information to try out our agent. We ask EPIKIT to trace its actions, and we call `interpret` on the name of our program. The database does not contain an empty disjunction; and so it cannot return just yet. By examining the sentences in the database and using the definition of `resolution` given earlier, it finds several possible deductions. It takes the first of these (involving the first two clauses in the `database` theory) and adds the consequence to the database. On the second step, this deduction is still valid but the conclusion is not added to the database again because it is already contained therein and, therefore, that action is bad. The next possible conclusion is, therefore, considered. The interpreter then finds and adds a third conclusion, viz. `(list 'or)`. On the fourth step, a `return` action becomes doable (due to the manifest contradiction), and the process halts.

```
User: (trace-action '$x)
Lisp: DONE

User: (interpret 'program-2)
Action: (CALL 'SAVE (LIST 'OR 'Q '(NOT R)) 'DATABASE)
Action: (CALL 'SAVE (LIST 'OR 'Q) 'DATABASE)
Action: (CALL 'SAVE (LIST 'OR) 'DATABASE)
Action: (RETURN TRUE)
Lisp: TRUE
```

Looking at the performance of our program, we may find that we really want something different. Let us say, for example, that we preferred to have the agent draw short conclusions before long ones (in the hope that a contradiction will be found faster that way). We can easily modify our program to do this by simply adding an additional axiom. The following sentence asserts that one conclusion is better than another if it is shorter.

```
User: (save '(<= (better (call 'save (list 'or @l) 'database)
User:                    (call 'save (list 'or @m) 'database))
User:           (< (list-length (list @l)) (list-length (list @m)))))
```

Having added this sentence, we can rerun our example. First, we reinitialize the theory. Then, we call `interpret` again. This time the first conclusion is bypassed in favor of the second conclusion, since the second conclusion is shorter.

```
User: (deftheory database
User:   (or p q)
User:   (or (not p) (not r))
User:   (or (not p))
User:   (or (not q)))
Lisp: DONE

User: (interpret 'program-2)
Action: (CALL 'SAVE (LIST 'OR 'Q) 'DATABASE)
Action: (CALL 'SAVE (LIST 'OR) 'DATABASE)
Action: (RETURN TRUE)
Lisp: TRUE
```

In similar fashion, it is possible to augment our program to include additional rules of inference (to implement nonmonotonic reasoning, for example). We can add additional constraints on conclusions (e.g. probabilities, branchiness, etc.). We can also make available additional actions (e.g. asking questions of the user, as in consultation systems).

Of course, the version of resolution shown here is not a particularly efficient implementation. However, it is easy to tune it up. For example, instead of searching linearly through the facts in the `database` theory, we could use EPIKIT's associative indexing scheme. We could cache possible conclusions, so they need not be recomputed on every step. And so forth.

6 Miscellaneous

In addition to its capabilties for database manipulation, inference, and partial program interpretation, EPIKIT provides a variety of useful subroutines for manipulating terms and sentences.

The `simp` command simplifies expressions using a variety of identities for common algebraic functions. The command is idempotent but the result is not necessarily canonical.

```
User: (simp '(/ (* 2 a (expt a 2)) (+ 1 1)))
Lisp: (EXPT A 3)
```

The `expand` command writes polynomials in fully expanded form. Unlike simplified form, expanded form *is* canonical (at least for polynomials). Consequently, the resulting value can be used in testing for polynomial equality.

```
User: (expand '(expt (+ a b) 2))
Lisp: (+ (EXPT A 2) (* 2 A B) (EXPT B 2))
```

The `reverse-rule` command reverses the direction of implications – forward to backward and backward to forward. `skolemize` eliminates all occurrences of quantifiers in the sentence supplied as argument. Universal quantifiers are dropped, and existential quantifiers are replaced by suitable Skolem constants and functions. `booleanize` converts logical operators to the three basic boolean operators `and`, `or`, and `not`. `cnf` converts its argument to conjunctive normal form, and `dnf` converts its argument to disjunctive normal form.

```
User: (reverse-rule '(<= (poisonous $x) (purple $x) (mushroom $x)))
Lisp: (=> (PURPLE $X) (MUSHROOM $X) (POISONOUS $X))

User: (skolemize '(exists $x (forall $y (exists $z (p $x $y $z)))))
Lisp: (P F1 $Y (F2 $Y))

User: (booleanize '(=> p q r))
Lisp: (OR (NOT P) (NOT Q) R)

User: (cnf '(exists $z (and (p $x) (not (and (q $y) (r $z))))))
Lisp: (AND (P $X) (OR (NOT (Q $Y)) (NOT (R (F3 $X $Y)))))

User: (dnf '(exists $z (and (p $x) (not (and (q $y) (r $z))))))
Lisp: (OR (AND (P $X) (NOT (Q $Y))) (AND (P $X) (NOT (R (F4 $X $Y)))))
```

Finally, there are matchers of varying degrees of power. `identp` succeeds if and only if its arguments are *identical* (with the condition that quoted lists and equivalent lists of quoted subexpressions are treated as identical). `samep` succeeds if and only if its two arguments are identical after consistent variable renaming. `matchp` succeeds if and only if the second expression is an instance of the first. `mgup` succeeds if and only if the two arguments unify. `unifyp` succeeds if and only if the two arguments unify after consistent renaimng of the variables of one of the arguments.

```
User: (identp '(p a '(g b) c) '(p a (list 'g 'b) c))
Lisp: T

User: (identp '(p $x b) '(p $y b))
Lisp: NIL

User: (samep '(p $x b) '(p $y b))
Lisp: T

User: (samep '(p $x b) '(p a b))
Lisp: NIL

User: (matchp '(p $x b) '(p a b))
Lisp: T

User: (matchp '(p $x b) '(p a $y))
Lisp: NIL

User: (mgup '(p $x b) '(p a $y))
Lisp: T

User: (mgup '(p $x b) '(p a $x))
Lisp: NIL

User: (unifyp '(p $x b) '(p a $x))
Lisp: T
```

EPIKIT also includes a full range of utilities (e.g. secondary storage handling) and debugging aids (e.g. tracing of all inference routines).

7 Conclusion

Please remember that this document is just an introduction to EPIKIT. It is intended to be suggestive, not exhaustive. Although the subroutines presented here illustrate the key features of EPIKIT, there are many other subroutines as well. Although the examples used here are quite simple (in keeping with the introductory nature of the presentation), EPIKIT is capable of and intended for use in the construction of very complex knowledge-based systems.

To this end, Epistemics is working on a variety of other products coordinated with EPIKIT. Epilibrarian is a system that helps programmers in building KIF libraries. Epireasoner is a theorem proving system. Epiprogrammer is a system that helps programmers in building Common Lisp programs (usually involving specifications written in KIF).