Resolution Theorem Prover

Background

A Resolution Theorem Prover (RTP) is presented. The RTP can prove a statement by contradiction with respect to an First-Order Logic (FOL) axiom set represented in Conjunctive Normal Form (CNF), known as the Knowledge Base (KB). All inferences are done by logical resolution.

Since ancient times, philosophers have been aware of modus ponens and model tollens, which both work from an implication statement (e.g.: A=>B). If you know A to be true, you can infer B. If you know B to be false, you can infer -A (where "-x" means 'NOT x'). Logical Resolution (LR) is a more recent discovery. LR allows inferences of the type:

$$(A \vee B) \wedge (-B \vee C) \Rightarrow (A \vee C)$$

Such inferences prove powerful because they are able to do a complete search on a given axiom set. If a given statement can be disproven (by contradiction) against an axiom set, resolution will do it.

The big idea is that a given statement A can be proven against a KB if one adds -A to the KB, and repeatedly applies resolution to various statements in the KB until a one results in a nil. Once -A is disproven, A is proven. More discussion can be found in Russel/Norvig on pg. 306 and following. The following discussion concerns the implementation.

The RTP presented here begins by building a KB object from a file. The complement of the statement to be proven is then added to the KB. The resolution function then chooses the first statement and begins to iterate over the other statements looking for one that will resolve with it. Once one is found, a resolution is produced, resulting in a new statement being added to the KB. The resolution function is recursive, so each call instantiates a new copy of the original KB plus the resolution statement. If a given call to the resolution function results in a logical NIL, the original statement has been proven. Instead of returning nil, the function returns the KB object, which now contains all that is necessary to reconstruct the proof. If, however, the resolution function finds something other than nil, it will return nil, which tells parent callers that this particular search branch was unfruitful.

Building the Knowledge Base

The original KB is held in a file and is written implicitly in CNF. Each statement is written on one line, and each clause is separated by a space, which is understood to be an 'OR'. Each statement is either positive or negative (have a "-" before it to indicate 'NOT'). Each statment is made up of a name, and a set of parameters in parenthesis. All elements are parsed as strings and kept as strings for all later manipulation. Even variables (which are signified by a "?" followed by an integer), are processed and manipulated in the Lisp code as strings. The lines in a KB file look like the following:

```
-isHuman(?1) mortal(?1)
-mortal(?3) -yearsSinceBirth(?3,?1) -greaterThan(?1,150) isDead(?3,?2)
```

As each clause is parsed, and Clause object is created, storing the parameters and whether or not it is negated. For simplicity's sake, any named Clause must always have the same number of parameters.

Once a full statement has been parsed, the list of Clause objects are slotted into a Statement object. As each statement is parsed it is stored in a KnowledgeBase object. The KB object stores various other information as well, such as the highest named variable and all resolutions so far. The highest named varible is used to make the variables in each new statement unique. As each statement is added, the number of each variable is bumped up to make them different from any preexisting variables. This process is also done on each resolution. Before the output of a resolution can be added to the KB, its variables are made unique. Such can be seen in the proof trees (later).

Effecting a Proof

After a KB is instantiated and populated, a statement to be proven can be generated. The 'prove function accepts that statement and the KB as parameters. The complement of the statement is then added to the KB, and the search for resolution begins. The resolution function (resolveToKB) will recursively search not only through all possible sets of statements to resolve, but all possible resolutions between those statements-- applicable where two statements may resolve in more than one way. The KB object stores the proof tree, which grows as the recursive search deepens, and then prunes back when the algorithm backs up. This action is implicit in the program structure because each KB object is copied anew when the resolution function is called recursively. All information necessary to reconstruct a proof tree is stored in the KB object, so it is the obvious object to return upon a

successful proof by contradiction.

The Code

The attached code is separated into the following files:

main.lsp	(main file)
/home/graham/work/lisp/lib/util.lsp	(general functions)
util.lsp	(functions useful here)
classes.lsp	(class definitions)
methods.lsp	(object methods, and some functions)

In order to run the code, look at "main.lsp". There are a series of runs laid out. In each case, a 'let statement opens a scope for a KB that is built from a file, then one attempt is made to prove a statement. The prove function will report whether or not the proof was successful. If successful, it will print out a proof tree.

Domains

Three axiom sets were built for this project. The first is "marcus.cnf". It contains statements similar to those presented in class. They are slightly different here to match this particular implementation. Against this set the following two statements are attempted:

1. isDead(Marcus,1969)
2. -isDead(Marcus,1969)

The year 1969 refers to the number of years since Marcus' birth. The first statement was proven, but the second statement was not. The output from this and the other proofs can be seen in the appendices.

The next axiom set concerns butterflies. After parsing a number of statements about butterflies and insects in general, several statements are also made about two specific butterflies named Bubba and Chuck. Attempts are made to prove that these are or are not Monarchs.

Finally, a generic axiom set was built merely to show that the program can handle greater levels of complexity. The set consists of a large number of statements, each of which contains several Clauses (named A,B,C,D), that have four parameters each, some variables and some constants. An

attempt was made to build an arbitrarily large set to push the limits of the program. Once the set had been built, a test was put forth-- to prove Z(Bubba). Since no statement in the KB includes Z (or even Bubba), there should be no way to prove Z(Bubba). However, if the original KB is inconsistent, we will be able to prove Z(Bubba), because you can prove anything with an inconsistent KB. Using this test, statements were removed until the KB could not prove Z(Bubba). Then an attempt was made to prove something logical. Results are in the appendices.

APPENDICES

The following pages contain:

1. code
2. KB loading files
3. output from "main.lsp"
    - various attempted proofs
    - each KB after it has been loaded
    - a proof tree for each successful proof

```
;; ################################
;; ######  FILE: main.lsp  #########
;; ################################
;;
;; Class definitions

;; Inclusions
(load "/home/graham/work/lisp/lib/util.lsp")
(load "util.lsp")
(load "classes.lsp")
(load "methods.lsp")


;; MARCUS AXIOM SET
(let ((kb (make-instance 'KnowledgeBase :file "marcus.cnf" )))
  (printKnowledgeBase kb)
  (prove kb (make-instance 'Statement :knowledgeBase kb :line "isDead(Marcus,1969)" )))

(let ((kb (make-instance 'KnowledgeBase :file "marcus.cnf" )))
  (prove kb (make-instance 'Statement :knowledgeBase kb :line "-isDead(Marcus,1969)" )))


;; BUTTERFLY AXIOM SET
(let ((kb (make-instance 'KnowledgeBase :file "butterflies.cnf" )))
  (printKnowledgeBase kb)(q)
  (prove kb (make-instance 'Statement :knowledgeBase kb :line "isMonarch(Bubba)" )))

(let ((kb (make-instance 'KnowledgeBase :file "butterflies.cnf" )))
  (prove kb (make-instance 'Statement :knowledgeBase kb :line "-isMonarch(Chuck)" )))

(let ((kb (make-instance 'KnowledgeBase :file "butterflies.cnf" )))
  (prove kb (make-instance 'Statement :knowledgeBase kb :line "isMonarch(Chuck)" )))


;; GENERIC AXIOM SET
(let ((kb (make-instance 'KnowledgeBase :file "generic.u.cnf" )))
  (printKnowledgeBase kb)
  (prove kb (make-instance 'Statement :knowledgeBase kb :line "Z(Bubba)" )))
(sleep 1)

(let ((kb (make-instance 'KnowledgeBase :file "generic.u.cnf" )))
  (prove kb (make-instance 'Statement :knowledgeBase kb :line "A(Jane,Tom,Dick,Harry)" )))
```

```
;; ################################
;; ######  FILE: classes.lsp  ######
;; ################################
;;
;; Class definitions


;; ####################
;; ### CLASS Clause  ###
;; ####################
;;
;; Represents a clause in FOL
;;
;; For this project, all clauses will be written such that the text of each clause is unique on its
;; vars.  Therefore, to determine if two clauses can be resolved, one need only verify that the text
;; matches and the truth does not-- one must be the "not" of the other.  Another convention will also
;; be followed in this project: any time or location or other oft-used variables will comprise one
;; letter and a number (could be zero).  Such variables will be kept unique by changing the variable
;; names to clauses as they are added to the knowledge base.
;;
;;
(defclass Clause ()
  ((statement    :accessor getStatement      ;; pointer to parent Statement
                 :initarg :statement
                 :initform nil)
   (topvar       :accessor getTopvar         ;; number of topvar
                 :initarg :topvar
                 :initform nil)
   (word         :accessor getWord           ;; word used to build clause
                 :initarg :word
                 :initform nil)
   (text         :accessor getText           ;; name or text of clause (a string)
                 :initarg :text
                 :initform nil)
   (truth        :accessor getTruth          ;; t or nil(representing a "not")
                 :initarg :truth
                 :initform t)
   (parameters   :accessor getParameters     ;; ordered list of parameters (strings)
                 :initarg :parameters
                 :initform nil)))

;; extra Args: <string> <int>
;; Effect: initializes Clause by parsing word, any vars must be named higher than topvar
(defmethod initialize-instance :after ((c Clause) &rest args)
  (if (< 0 (length (getWord c)))
```

```
      (let ((word (getWord c)))
        (if (equal #\- (char word 0))
            (setf (getTruth c) nil))
        (setf (getText c) (textOf word))
        (setf (getParameters c) (parametersOf word (getTopvar c))))
      (checkParameters c)))


;; ########################
;; ### CLASS Statement  ###
;; ########################
;;
;; Represents a statement in FOL
;;
;;
(defclass Statement ()
  ((knowledgeBase   :accessor getKnowledgeBase     ;; pointer to KnowledgeBase
                    :initarg :knowledgeBase
                    :initform nil)
   (line            :accessor getLine              ;; pointer to line
                    :initarg :line
                    :initform nil)
   (clauses         :accessor getClauses           ;; pointer to clauses
                    :initarg :clauses
                    :initform '())))

;; Effect: initializes Statement with clauses by parsing string
(defmethod initialize-instance :after ((s Statement) &rest args)
  (if (getLine s)
      (let ((topvar (getTopvar (getKnowledgeBase s))))          ;; get highest var name
        (dolist (word (split (getLine s)) nil)
          (let ((newClause (make-instance 'Clause
                                          :word word
                                          :topvar topvar
                                          :statement s)))
            (if (getParameters newClause)
                (pushEnd (getClauses s) newClause))))))
  (setf (getTopvar (getKnowledgeBase s)) (highestVar s)))       ;; set new "top" var number



;; ############################
;; ### CLASS KnowledgeBase ###
;; ############################
;;
```

```
;; Represents a knowledge base, the basic unit of which is a statement or a variable binding.
;; Each statement is a list of clauses in Conjunctive Normal Form (CNF).  Each variable binding is a
dotted pair (variable,
;; value)
;;
;; ###  EXTERNAL METHODS  ###
;;
;;
(defclass KnowledgeBase ()
  ((statements   :accessor getStatements     ;; pointer to statements
                 :initarg :statements
                 :initform '())
   (file         :accessor getFile           ;; file containing CNF statements
                 :initarg :file
                 :initform nil)
   (topvar       :accessor getTopvar         ;; int value of top named var, where each var is of
                 :initarg :topvar            ;;   the format "?<int>"
                 :initform 0)
   (tree         :accessor getTree           ;; proof tree (stored as a list of strings)
                 :initarg :tree
                 :initform '())
   (path         :accessor getPath           ;; Tuples of past statments/clauses used for resolution.
                 :initarg :path              ;;   This list used to prevent infinite loops
                 :initform '())
   (level        :accessor getLevel          ;; recursion level
                 :initarg :level
                 :initform 0)))


;; Effect: initializes KnowledgeBase by reading Statements from a file
(defmethod initialize-instance :after ((kb KnowledgeBase) &rest args)
  (if (getFile kb)
      (setf (getStatements kb) (readStatements (getFile kb) kb))))
```

```
;; ################################
;; ######  FILE: methods.lsp  ######
;; ################################
;;
;; Various methods


;; Args: KnowledgeBase
;; Returns: local copy of KB
(defmethod copyKnowledgeBase ((kb KnowledgeBase))
  (let ((kb2 (make-instance 'KnowledgeBase
                            :level (getLevel kb)        ;; recursion depth
                            :topvar (getTopvar kb))))   ;; initialize local copy
    (setf (getStatements kb2) (copyStatements kb2 (getStatements kb))) ;; set copy of statements
    (setf (getTree kb2) (copy-list (getTree kb)))                      ;; set copy of proof tree
    (setf (getPath kb2) (copy-list (getPath kb)))                      ;; set copy of resolution path
    (return-from copyKnowledgeBase kb2)))


;; Args: KnowledgeBase
;; Effect: prints all Statements to STDOUT
(defmethod printKnowledgeBase ((kb KnowledgeBase) &rest args)
  (dolist (s (getStatements kb) nil)
    (printStatement s))
  t)       ;; Returns t

;; Args: null KnowledgeBase
(defmethod printKnowledgeBase (nullKB &rest args)
  (format t "Null KnowledgeBase~%")
  nil)     ;; Returns nil


(defmethod printTree ((kb KnowledgeBase))
  (dolist (line (getTree kb) nil)
    (format t line)
    (format t "~%"))
  (format t "~%"))


;; Args: file
;; Returns: list of Statements
(defun readStatements (file kb)
  (let ((statements '()))
    (dolist (line (readFile file) statements)
      (if (< 0 (length line))
```

```
          (let ((newStat (make-instance 'Statement :line line :knowledgeBase kb)))
            (pushEnd statements newStat)
            (setf (getTopvar kb) (highestVar newStat)))))))


;; Args: Statement KnowledgeBase
;; Returns: local copy of Statement
(defmethod copyStatement ((s Statement) (kb KnowledgeBase))
  (let ((c2s '())                                   ;; new clauses
        (s2 (make-instance 'Statement               ;; initialize local copy
                           :knowledgeBase kb)))
    (dolist (c (getClauses s) nil)                  ;; iterate over clauses
      (let ((newClause (copyClause c s2)))
        (pushEnd c2s newClause)))              ;; collect clause copies
    (setf (getClauses s2) c2s)                      ;; set new clause list in s2
    (return-from copyStatement s2)))


;; Args: KnowledgeBase, list of Statements
;; Returns: local copy of Statement
(defmethod copyStatements ((kb KnowledgeBase) statements)
  (let ((s2s '()))                              ;; new statements
    (dolist (s statements nil)                  ;; iterate over preexisting statements
      (let ((newStatement (copyStatement s kb)))
        (pushEnd s2s newStatement)))      ;; collect statement copies
    (return-from copyStatements s2s)))


;; Args: Statement
;; Effect: prints Statement to STDOUT
(defmethod printStatement ((s Statement))
  (format t "    ")
  (printClause (car (getClauses s)))
  (dolist (c (cdr (getClauses s)) nil)
    (format t " OR ")
    (printClause c))
  (format t "~%"))


;; Args: list of Statements
;; Effect: prints each Statement to STDOUT
(defun printStatements (statements)
  (dolist (s statements nil)
    (printStatement s)))
```

```
;; Args: Statement
;; Returns: text describing statement
(defmethod statementText ((s Statement))
  (let ((line "")
        (clause1 (car (getClauses s)))
        (otherClauses (cdr (getClauses s))))
    (setq line (concatenate 'string line (clauseText clause1)))
    (dolist (c otherClauses nil)
      (setq line (concatenate 'string line " OR " (clauseText c))))
    (return-from statementText line)))


(defmethod uniqueVars ((kb KnowledgeBase) (s Statement))
  (lowestVarsPossible s)
  (setf (getKnowledgeBase s) kb)                          ;; set kb in slot in s
  (let ((topvar (getTopvar kb)))                          ;; get highest var name in kb
    (dolist (c (getClauses s) nil)                        ;; iterate over clauses in s
      (incrementClause c topvar)))                        ;; make vars unique
  (setf (getTopvar kb) (highestVar s))  ;; set new "top" var number in kb
  (return-from uniqueVars s))


;; Args: Statement
;; Returns: highest int in var name in this Statement
(defmethod highestVar ((s Statement))
  (let ((highest 0)
        (orig (getTopvar (getKnowledgeBase s))))
    (dolist (clause (getClauses s) nil)
      (dolist (par (getParameters clause) nil)
        (if (isVariable par)
            (if (> (numberOf par) highest)
                (setq highest (numberOf par))))))
    (if (< orig highest)
        (return-from highestVar highest)
        (return-from highestVar orig))))



;; Args: Statement
;; Returns: highest int in var name in this Statement
(defmethod lowestVar ((s Statement))
  (let ((lowest 999))
    (dolist (c (getClauses s) nil)
      (dolist (p (getParameters c) nil)
```

```
        (if (isVariable p)
            (if (< (numberOf p) lowest)
                (setq lowest (numberOf p))))))
    (return-from lowestVar lowest)))



;; Args: Statement
;; Effect: lowers var values to lowest possible
(defmethod lowestVarsPossible ((s Statement))
  (let ((dec (- 1 (lowestVar s))))
    (dolist (c (getClauses s) nil)
      (incrementClause c dec))))



;; Unify and Resolve two statements, if possible
;; Args: Statement1, Statement2
;; Returns: list of Statements, each representing one possible resolution
(defmethod unisolve((kb KnowledgeBase) (s1 Statement) (s2 Statement))
  (let ((resolutions '())     ;; list of dotted pairs containing statements and associated bindings
        (i -1)
        (j -1))
    (dolist (ci (getClauses s1) nil)            ;; loop over clauses in s1
      (incf i)
      (setq j -1)
      (dolist (cj (getClauses s2) nil)          ;; loop over clauses in s2
        (incf j)
        (if (resolvableClauses ci cj)
            (if (notInPath kb s2 j)
                (let ((result (resolve kb s1 i s2 j)))
                  (appendPath kb s2 j)
                  (pushEnd resolutions result))))))
    (return-from unisolve resolutions)))



;; Args: Statement
;; Returns: number of clauses
(defmethod statementLength ((s Statement))
  (length (getClauses s)))



;; Args: Statement1, index1, Statement2, index2
;; Returns: a new Statement created by resolving s1 with s2, which was done by removing
;;   clause i from s1, and clause j from s2, which were unified as negations of each other
```

```
;;   The other clauses in s1 were connected by OR with the clauses from s2.  The unified
;;   vars from i,j were unified throughout s1,s2
;; Effect: Should not have effects (i.e. Shouldn't alter tree or path)
(defmethod resolve ((kb KnowledgeBase) (s1 Statement) i (s2 Statement) j)
  (let ((replace (make-hash-table :test 'equal))  ;; hash of vars to be replaced
        (s1b (copyStatement s1 kb))               ;; local copy of s1
        (s2b (copyStatement s2 kb)))              ;; local copy of s2
    (let ((c1s (getClauses s1b))                  ;; clauses from s1b
          (c2s (getClauses s2b)))                 ;; clauses from s2b
      (let ((n (length c1s))                      ;; number of clauses in s1b
            (c1 (nth i c1s))                      ;; clause i from s1b
            (c2 (nth j c2s)))                     ;; clause j from s2b
        (let ((p1s (getParameters c1))            ;; parameters from c1
              (p2s (getParameters c2))            ;; parameters from c2
              (m (length (getParameters c1))))    ;; number of parameters in c1 (must be same as c2)
          (loop for k from 0 to (1- m) do         ;; loop over each parameter of the two clauses
               (let ((p1 (nth k p1s))             ;; parameter k from c1
                     (p2 (nth k p2s)))            ;; parameter k from c2
                 (cond
                   ((isVariable p2) (setf (gethash p2 replace) p1))  ;; p2 to be replaced with p1
                   ((isVariable p1) (setf (gethash p1 replace) p2))  ;; p1 to be replaced with p2
                   (t )))))                       ;; PRE: if both constants, both must already be
equivalent
                                                  ;;   (as checked in 'resolvableParameters)
        (let ((newClauses (append (subseq c1s 0 i)
                                  (subseq c1s (1+ i))
                                  (subseq c2s 0 j)
                                  (subseq c2s (1+ j)))))
            (resolution (make-instance 'Statement  ;; initialize resolution statement
                                       :knowledgeBase (getKnowledgeBase s1))))
          (setf (getClauses resolution) (copyClauses resolution newClauses))
          (replaceVars resolution replace)        ;; Do var replacements called for by resolution
          (return-from resolve (cons resolution replace)))))))


;; Args: Statement
;; Returns: Boolean (T if any two clauses resolve to T)
(defmethod selfResolves ((s Statement))
  (let ((i -1)
        (clausesToRemove '()))
    (dolist (ci (getClauses s) nil)
      (incf i)
      (let ((j -1))
        (dolist (cj (getClauses s) nil)
          (incf j)
```

```
          (if (not (eql i j))
              (if (resolvableClauses ci cj)
                  (return-from selfResolves t)))))))
  nil)

;                  (progn
;                    (push i clausesToRemove)
;                    (push j clausesToRemove)))))))
;    (let ((clauses '())
;          (i -1))
;      (dolist (c (getClauses s) nil)
;        (incf i)
;        (if (not (member i clausesToRemove))
;            (pushEnd clauses c)))
;      (setf (getClauses s) clauses))))


;; Args: Statement, <hash>
;; Effect: Replaces each var with its corresponding hash value
(defmethod replaceVars ((s Statement) replace)
  (dolist (c (getClauses s) nil)
    (let ((newPars '()))
      (dolist (p (getParameters c) nil)
        (if (gethash p replace)
            (pushEnd newPars (gethash p replace))
            (pushEnd newPars p)))
      (setf (getParameters c) newPars))))          ;; replace old pars with new


;; Args: Statement1 Statement2
;; Returns: Boolean
(defmethod resolvableStatements ((s1 Statement) (s2 Statement))
  (dolist (c1 (getClauses s1) nil)
    (dolist (c2 (getClauses s2) nil)
      (if (resolvableClauses c1 c2)
          (return-from resolvableStatements t)
          ;(format t "NOT RESOLVABLE: ~S <-> ~S~%" (clauseText c1) (clauseText c2))
          )))
  nil)


;; Args: Statement1, list of statements
;; Returns: list of statements resolvable with s1
(defmethod justResolvableStatements ((s Statement) statements)
  (let ((rStatements '()))
```

```
    (dolist (si statements nil)
      (if (resolvableStatements s si)
          (pushEnd rStatements si)))
    (return-from justResolvableStatements rStatements)))


;; Args: Clause Statement
;; Returns: content-wise copy of Clause
(defmethod copyClause ((c Clause) (s Statement))
  (let ((c2 (make-instance 'Clause
                           :statement s
                           :topvar (getTopvar (getKnowledgeBase s))
                           :text (getText c)
                           :truth (getTruth c))))
    (setf (getParameters c2) (getParameters c))  ;; set pars here to prevent renumbering
    (return-from copyClause c2)))


;; Args: Clause Statement
;; Returns: content-wise copy of Clause
(defmethod copyClauses ((s Statement) clauses)
  (let ((clausesCopy '()))
    (dolist (c clauses nil)
      (let ((newClause (copyClause c s)))
        (pushEnd clausesCopy newClause)))
    (return-from copyClauses clausesCopy)))


;; Args: Clause
;; Effect: prints Clause to STDOUT
(defmethod printClause ((c Clause))
  (if (not (getTruth c))
      (format t "-"))
  (format t (getText c))
  (format t "(")
  (format t (car (getParameters c)))
  (dolist (par (cdr (getParameters c)) nil)
    (format t ",")(format t par))
  (format t ")"))

(defmethod printClause (nullClause)
  (format t "Null Clause"))


;; Args: Clause
```

```
;; Returns: text describing clause
(defmethod clauseText ((c Clause))
  (let ((line ""))
    (if (not (getTruth c))
        (setq line "-"))
    (setq line (concatenate 'string line
                            (getText c)
                            "("
                            (car (getParameters c))))
    (dolist (par (cdr (getParameters c)) nil)
      (setq line (concatenate 'string line "," par)))
    (setq line (concatenate 'string line ")"))
    (return-from clauseText line)))

(defmethod clauseText (c)
  ())


;; Args: Clause <int>
;; Effect: increments vars in Clause by topvar
(defmethod incrementClause ((c Clause) topvar)
  (let ((newPars '()))
    (dolist (par (getParameters c) nil)
      (if (isVariable par)
          (let ((newPar (increment par topvar)))
            (pushEnd newPars newPar))
          (pushEnd newPars par)))
    (setf (getParameters c) newPars)))


;; Args: Clause1, Clause2
;; Returns: Boolean
(defmethod resolvableClauses ((c1 Clause) (c2 Clause))
  (and (equal (getText c1) (getText c2))           ;; resolvable only if text equal
       (not (equal (getTruth c1) (getTruth c2)))   ;; resolvable only if truth not equal
       (resolvableParameters c1 c2)))              ;; resolvable only if pars resolvable


;; may not be necessary
;; Args: Clause1, Clause2 (all parameters must be constants)
;; Returns: Boolean
(defmethod resolvableClauses-constant ((c1 Clause) (c2 Clause))
  (if (and (allParametersConstants c1)
           (allParametersConstants c2))
      (resolvableClauses c1 c2)
```

```
      nil))


;; Args: Clause
;; Returns: Boolean
(defmethod allParametersConstants ((c Clause))
  (dolist (p (getParameters c) nil)
    (if (isVariable p)
        (return-from allParametersConstants nil)))
  t)


;; Args: Clause1, Clause2
;; Returns: Boolean
;; PRE: the text of both Clauses match
(defmethod resolvableParameters ((c1 Clause) (c2 Clause))
  (checkParameters c1)
  (checkParameters c2)
  (let ((n (length (getParameters c1)))
        (pars1 (getParameters c1))
        (pars2 (getParameters c2)))
    (loop for i from 0 to (1- n) do
         (if (not (or (or (isVariable (nth i pars1))         ;; resolvable if either par
                          (isVariable (nth i pars2)))        ;;   is a var
                      (equal (nth i pars1) (nth i pars2))))  ;; pars equal
             (return-from resolvableParameters nil))))       ;; if neither true, return nil
  t)  ;; otherwise return t



;; Args: KnowledgeBase, Statement
;; Effect: if s can be proven by contradiction, prints resulting KB
(defmethod prove ((kb KnowledgeBase) (s Statement))
  (format t "~%TRYING TO PROVE THAT :")(printStatement s)(format t "~%")

  ;; Add complement of S to KB
  ;;   Since the complement returns a list of anded statements that may have shared parameters,
  ;;   the complement can only be added to the KB when using constants
  (dolist (sb (complement-statement s) nil)          ;; 'complement returns a list of statements
    (let ((sc (uniqueVars kb sb)))                   ;; set unique var numbers
      (push sc (getStatements kb))))                ;; add s2 to knowledge base

  ;; Use each statement to try attempt resolution
  (dolist (s3 (getStatements kb) nil)                ;; iterate through all statements
    (let ((resultKB (resolveToKB kb s3)))            ;; resolve with respect to knowledge base
```

```
      (if resultKB                                ;; if returns a KB, contradiction was found
          (progn
            (printTree resultKB)                  ;; print proof tree
            (format t "THEREFORE :")(printStatement s)(format t "~%")
            (return-from prove t)))))

  (format t "COULD NOT PROVE THAT :")(printStatement s)(format t "~%")
  nil)


;; Args: KnowledgeBase Statement
;; Effect: appends text of statement to the proof tree
(defmethod appendTree ((kb KnowledgeBase) (s Statement))
  (let ((line (statementText s)))
    (pushEnd (getTree kb) line)))

;; Args: KnowledgeBase <string>
;; Effect: appends string to the proof tree
(defmethod appendTree ((kb KnowledgeBase) line)
  (pushEnd (getTree kb) line))


;; Args: KnowledgeBase Statement <int>
;; Effect: appends to resolution path information for use in preventing inf. loops
(defmethod appendPath ((kb KnowledgeBase) (s Statement) j)
  (push (cons j (statementText s)) (getPath kb)))


;; Args: KnowledgeBase Statement <int>
;; Returns: Boolean
(defmethod notInPath ((kb KnowledgeBase) (s Statement) j)
  (dolist (p (getPath kb) nil)
    (if (and (equal j (car p))      ;; indices are equal
             (equal (statementText s) (cdr p)))
        (return-from notInPath nil)))  ;; Return nil
  t)                                   ;; Else t


;; Args: KnowledgeBase Statement
;; Effect: appends text of statement to the proof tree
(defmethod appendTreePreamble ((kb KnowledgeBase) (s Statement) preamble)
  (let ((line (concatenate 'string preamble (statementText s))))
    (pushEnd (getTree kb) line)))
```

```
;; Recursive function to do depth-first search. Each level passes a locally-modified KB. When a
;;   resolution returns nil, s has been proven
;; Args: KnowledgeBase, Statement, branch(text to be added to tree)
;; Returns: a KnowledgeBase, or nil if never resolves
;; Effect: will append to tree and path, but only once in each call
(defmethod resolveToKB ((kb KnowledgeBase) (s Statement) &optional branch)
  (let ((kb2 (copyKnowledgeBase kb)))                    ;; make local copy of KB
    (incf (getLevel kb2))
    (if branch
        (appendTree kb2 branch))
    (appendTree kb2 s)
    ;(format t "~%############ >>> SO FAR (~S):~%" (getLevel kb2))(printTree kb2)


    (if (< 99 (getLevel kb2))              ;; prevent deep recursion as it usually indicates an
infinite loop
        (return-from resolveToKB nil))


    (let ((statements (justResolvableStatements s (getStatements kb2))))  ;; statements in KB2
resolvable with s
      (dolist (si statements nil)                          ;; iterate over resolvable statements
        (dolist (result (unisolve kb2 s si) nil)           ;; iterate over resolutions
          (let ((sj (car result))
                (replace (cdr result)))
            (if (selfResolves sj)                          ;; IF a pair of clauses resolve to T, we
know it
                (return-from resolveToKB nil))             ;;   can never be NIL
            (let ((newStatement (uniqueVars kb2 sj)))
              (pushEnd (getStatements kb2) newStatement))  ;; append statement to kb2


            ;; IF resolution produced NIL
            (if (null (getClauses sj))                     ;; if no clauses left, proof by
contradiction successful
                (progn
                  (let ((branch2 (makeBranch si replace)))
                    (appendTree kb2 branch2))
                  (appendTree kb2 "     NIL")
                  (return-from resolveToKB kb2)))


            ;; ELSE apply recursively
            (let ((branch2 (makeBranch si replace)))
              (let ((resultKB (resolveToKB kb2 sj branch2)))       ;; take kb2,sj and apply
recursively
                (if (not (null resultKB))
                    (return-from resolveToKB resultKB)))))))))


  nil)
```

```
;; Args: Statement <hash-table>
;; Returns: <string>
(defmethod makeBranch ((s Statement) replace)
  (concatenate 'string
               "  \\~%"                   ;; formatting
               "   \\    / ~T"
               (statementText s)         ;; 2nd statement in resolution
               "~%"
               (hashBindings replace)     ;; bindings
               "~%     \\/"))



;; Check parameters for nulls
;; Args: Clause
;; Effect: errors out if null
(defmethod checkParameters ((c Clause))
  (dolist (p (getParameters c) nil)
    (if (null p)
        (progn
          (format t "ERROR: null parameter for:")
          (printStatement (getStatement c))
          (q)))))



;; Args: Statement
;; Returns: a list of statements. Each single statment is a set of OR'd clauses,
;;     and there is an implicit AND between each statement.
;;     Since the complement returns a list of anded statements that may have shared parameters,
;;     the complement can only be added to the KB when using constants
(defmethod complement-statement ((s Statement))
  (let ((kb (getKnowledgeBase s)))
    (let ((s2 (copyStatement s kb)))
      (if (equal 1 (length (getClauses s)))                    ;; IF: s2 only has one clause
          (let ((c (complement-clause (car (getClauses s2)))))  ;;     get complement of only clause
            (setf (getClauses s2) `(,c))
            (return-from complement-statement `(,s2)))          ;;     return list with one statement
          (let ((anded '()))                                   ;; ELSE: s2 has multiple clauses
            (dolist (c (getClauses s2) nil)
              (let ((c2 (complement-clause (copyClause c s2)))) ;;     complement of each clause
                (pushEnd anded c2)))                           ;;     complements implicitly AND'd
(DeMorgan)
            (format t "500 Statements: ")(printStatements anded)
            (return-from complement-statement anded))))))
```

```
;; Args: Clause
;; Returns: logically "notted" version of original clause
(defmethod complement-clause ((c Clause))
  (let ((c2 (copyClause c (getStatement c))))
    (let ((truth (not (getTruth c2))))     ;; complement(P) = not(P)
      (setf (getTruth c2) truth))
    return-from complement-clause c2))
```

```
;; ##############################
;; ######  FILE: util.lsp  ######
;; ##############################
;;
;; Functions of general use to this project


;; QUIT
(defun q () (quit))



;; From: http://cl-cookbook.sourceforge.net/strings.html
(defun split (line)
  (loop for i = 0 then (1+ j)
     as j = (position #\Space line :start i)
     collect (subseq line i j)
     while j))


;; Args: <string>
;; Returns: subseq up to the first paren
(defun textOf (word)
  (let ((j (position #\( word :start 0)))
    (if (equal #\- (char word 0))
        (subseq word 1 j)
        (subseq word 0 j))))


;; Args: <string> in format "name(par1,par2,...)"
;; Returns: list of parameters within parenthesis
(defun parametersOf (word topvar)
  (let ((parlist (subseq word
                      (1+ (position #\( word :start 0))
                      (1- (length word)))))
    (let ((outPars '())
          (pars
           (loop for i = 0 then (1+ j)
              as j = (position #\, parlist :start i)
              collect (subseq parlist i j)
              while j)))
      (dolist (par pars outPars)
        (if (isVariable par)
            (pushEnd outPars (increment par topvar))
            (pushEnd outPars par)))
```

```
      (return-from parametersOf outPars))))


;; Args: <string> <int>
;; Returns: var name, but with the number incremented by topvar
(defun increment (var topvar)
  (let ((number (numberOf var)))
    (setq number (+ number topvar))
    (concatenate 'string (string #\?) (write-to-string number))))


;; Args: parameter (string)
;; Returns: Boolean (t if par begins with "?")
(defun isVariable (par)
  (if (equal #\? (char par 0))
      t  nil))

;; Args: var  (Assumes a variable of format ?<number>)
;; Returns: integer representing number at end
(defun numberOf (var)
  (parse-integer (subseq var 1 (length var))))


;; Args: file
;; Returns: list of lines
(defun readFile (file)
  (let ((lines '())
        (line nil))
    (with-open-file (stream file)
      (loop while (setq line (read-line stream nil)) do
           (if (< 0 (length line))
               (pushEnd lines line))))
    (return-from readFile lines)))

;; Args: <hash-table>
;; Returns: a string containing bindings represented
(defun hashBindings (replace)
  (let ((line "    \\  / ~T {"))            ;; add bindings to proof tree
    (maphash #'(lambda (x y)
                 (setq line (concatenate 'string line " " x "/" y)))
             replace)
    (setq line (concatenate 'string line " }"))
    line))
```

```
;; ############################################################
;; ######  FILE: /home/graham/work/lisp/lib/util.lsp  ######
;; ############################################################
;;
;; Functions of general use

(setf (ext:package-lock (find-package "COMMON-LISP")) nil)             ;; enables altering
(setf (ext:package-definition-lock (find-package "COMMON-LISP")) nil)  ;;   of (setf nthcdr)


;; Args: a list
;; Returns: random element
(defun randomChoice (lst)
  (if (zerop (length lst))
      nil
      (nth (random (length lst)) lst)))


;; Args: list
;; Returns: unsorted list
(defun randomSort (lst)
  (let ((len (length lst))    ;; length of lst
        (numAlist nil)        ;; alist associating list elements with random numbesr
        (sortedNumAlist nil)  ;; randomly sorted version of numAlist
        (out nil))            ;; unsorted version of lst
    (dolist (ele lst t)
      (push `(,ele . ,(random len)) numAlist))
    (setq sortedNumAlist
          (sort numAlist #'(lambda (x y) (< (cdr x) (cdr y)))  ;; sort alist by the random numbers
                ))
    (dolist (ele sortedNumAlist out)
      (push (car ele) out))))


;; Args: list
;; Returns: subsequence of <lst>
(defun randomSet (lst n)
  (decf n)
  (subseq (randomSort lst) 0 (+ 1 n)))


;; Args: setf nthcdr : to define new action for the setf macro
;;       val (new value),  n (index of cdr in question),  lst (list to be altered)
;; Effect: destructively alters <lst>
(defmethod (setf nthcdr) (val n lst)
```

```
  (loop while (> n 0) do
       (progn
         (decf n)
         (setq lst (cdr lst))))
  (setf (cdr lst) val))


;; Args: lst, args
;; Effect: pushes args onto end of lst
(defmacro pushEnd (lst x)
  `(cond
     ((null ,x) lst)
     ((null ,lst) (push ,x ,lst))
     (t (setf (cdr (last ,lst)) (list ,x))
        ,lst)))
```

MARCUS.CNF

```
-isHuman(?1) mortal(?1)
-mortal(?3) -yearsSinceBirth(?3,?1) -greaterThan(?1,150) isDead(?3,?2)
-isPompeian(?1) diedBefore(?1,80)
hasErupted(volcano,79)
greaterThan(1969,150)
isHuman(Marcus)
isPompeian(Marcus)
yearsSinceBirth(Marcus,1969)
```

BUTTERFILES.CNF

```
-isInsect(?1) -hasTwoWings(?1) -wasCaterpillar(?1) isButterfly(?1)
-isOrange(?1) -migrates(?1) -isButterfly(?1) isMonarch(?1)
isInsect(?1) -isButterfly(?1)
hasTwoWings(?1) -isButterfly(?1)
wasCaterpillar(?1) -isButterfly(?1)
isButterfly(?1) -isMonarch(?1)
isOrange(?1) -isMonarch(?1)
migrates(?1) -isMonarch(?1)
isInsect(Bubba)
hasTwoWings(Bubba)
wasCaterpillar(Bubba)
isOrange(Bubba)
migrates(Bubba)
isInsect(Chuck)
hasTwoWings(Chuck)
wasCaterpillar(Chuck)
-isOrange(Chuck)
migrates(Chuck)
```

GENERIC.CNF

A(?1,?2,Tom,Dick) B(?1,Dick,?2,Tom)
A(?1,Dick,Tom,?2) C(?1,Tom,?2,Dick)
A(?2,?4,Dick,Tom)
A(?2,?4,Tom,Dick) B(?2,Dick,?4,Tom)
A(?2,Dick,Jane,Tom)
A(?2,Dick,Tom,?4)
A(?2,Dick,Tom,Jane)
A(?2,Jane,Dick,Tom)
A(?2,Jane,Tom,Dick) B(?2,Tom,?1,Dick)
A(?2,Tom,?3,?4) B(?2,Tom,?4,Dick)
A(?2,Tom,?4,Dick)
A(?2,Tom,Dick,Jane) C(?2,Tom,Jane,Dick)
A(?4,Dick,Harry,Tom)
A(?4,Dick,Tom,Harry)
A(?4,Harry,Dick,Tom) B(?4,Harry,Tom,Dick)
A(?4,Tom,?2,Dick)
A(?4,Tom,Harry,Dick)
A(Dick,?1,?2,Tom)
A(Dick,?1,Tom,?2) C(Dick,?2,?1,Tom)
A(Dick,?2,?4,Tom)
A(Dick,?2,Jane,Tom)
A(Dick,?2,Tom,?1)
A(Dick,?2,Tom,?4) B(Dick,?2,Tom,Jane)
A(Dick,?4,?2,Tom)
A(Dick,?4,Harry,Tom)
A(Dick,?4,Tom,?2)
A(Dick,?4,Tom,Harry)
A(Dick,Harry,?4,Tom) C(Dick,Harry,Jane,Tom)
A(Dick,Harry,Tom,?4)
A(Dick,Harry,Tom,Jane)
A(Dick,Jane,?2,Tom)
A(Dick,Jane,Harry,Tom)
A(Dick,Jane,Tom,?2) B(Dick,Jane,Tom,Harry)
A(Dick,Tom,?1,?2)
A(Dick,Tom,?2,?1)
A(Dick,Tom,?2,?4)
A(Dick,Tom,?2,Jane)
A(Dick,Tom,?4,?2)
A(Dick,Tom,?4,Harry) C(Dick,Tom,Harry,?4)
A(Dick,Tom,Harry,Jane)
A(Dick,Tom,Jane,?2)
A(Dick,Tom,Jane,Harry)
A(Harry,?4,Dick,Tom)

A(Harry,?4,Tom,Dick)

A(Harry,Jane,Dick,Tom)

A(Harry,Jane,Tom,Dick) B(Harry,Tom,?4,Dick)

A(Harry,Tom,Dick,?4)

A(Harry,Tom,Dick,Jane) C(Harry,Tom,Jane,Dick)

A(Jane,Dick,Harry,Tom) C(Jane,Dick,Tom,Harry)

A(Jane,Harry,Dick,Tom)

A(Jane,Harry,Tom,Dick)

A(Jane,Tom,?2,Dick)

A(Jane,Tom,Harry,Dick) B(Tom,?2,?1,Dick)

A(Tom,?2,Dick,?1)

A(Tom,?4,?2,Dick)

A(Tom,?4,Dick,?2) C(Tom,?4,Dick,Harry)

A(Tom,?4,Harry,Dick)

A(Tom,Dick,?4,Harry)

A(Tom,Dick,Harry,?4)

A(Tom,Dick,Harry,Jane)

A(Tom,Dick,Jane,Harry)

A(Tom,Harry,?4,Dick)

A(Tom,Harry,Dick,?4)

A(Tom,Harry,Dick,Jane) B(Tom,Harry,Jane,Dick)

A(Tom,Jane,?2,Dick)

A(Tom,Jane,Dick,?2)

A(Tom,Jane,Dick,Harry)

A(Tom,Jane,Harry,Dick) D(?1,Dick,?2,Tom)

B(?1,Dick,Dick,?2) C(?1,Dick,Dick,Tom)

B(?1,Dick,Tom,?2)

B(?2,?1,Dick,Tom)

B(?2,Dick,?1,Tom) D(?2,Dick,?4,Tom)

B(?2,Dick,Dick,?4)

B(?2,Dick,Dick,Jane)

B(?2,Dick,Dick,Tom)

B(?2,Dick,Jane,Tom) C(?2,Dick,Tom,?4)

B(?2,Dick,Tom,Jane)

-B(?2,Tom,?4,Dick)

B(?4,?2,Dick,Tom)

B(?4,Dick,?2,Tom) D(?4,Dick,Dick,Harry)

B(?4,Dick,Dick,Tom)

B(?4,Dick,Harry,Tom)

B(?4,Dick,Tom,Harry)

B(?4,Tom,Dick,Harry)

B(Dick,?1,?1,Tom) C(Dick,?1,?2,Tom)

B(Dick,?2,?2,Jane)

B(Dick,?2,?2,Tom) D(Dick,?2,?4,Tom)

B(Dick,?2,Jane,Tom) D(Dick,?2,Tom,Jane)

B(Dick,?4,?4,Harry)

B(Dick,?4,?4,Tom) C(Dick,?4,Harry,Tom)

B(Dick,?4,Tom,Harry)

B(Dick,Jane,Harry,Tom)

B(Dick,Jane,Jane,Harry)

B(Dick,Jane,Jane,Tom) D(Dick,Jane,Tom,Harry)

B(Dick,Tom,?4,Harry) D(Dick,Tom,Jane,Harry)

B(Dick,Tom,Tom,Harry)

B(Dick,Tom,Tom,Jane)

B(Harry,?4,Dick,Tom) C(Harry,Dick,?4,Tom)

B(Harry,Dick,Jane,Tom)

B(Harry,Dick,Tom,?4) D(Harry,Dick,Tom,Jane)

B(Harry,Harry,Harry,Harry) D(Harry,Jane,Dick,Tom)

B(Harry,Tom,Dick,Jane) D(Jane,?2,Dick,Tom)

B(Jane,Dick,?2,Tom) D(Jane,Dick,Dick,Harry)

B(Jane,Dick,Dick,Tom)

B(Jane,Dick,Harry,Tom) C(Jane,Dick,Tom,Harry)

B(Jane,Jane,Jane,Jane)

B(Jane,Tom,Dick,Harry)

B(Tom,?2,Dick,Jane)

B(Tom,?4,Dick,Harry)

B(Tom,Dick,?1,?2)

B(Tom,Dick,?2,?4)

B(Tom,Dick,?2,Jane)

B(Tom,Dick,?4,Harry) C(Tom,Dick,Dick,?4)

B(Tom,Dick,Dick,Harry)

B(Tom,Dick,Dick,Jane) D(Tom,Dick,Harry,?4) C(Tom,Dick,Harry,Jane)

B(Tom,Dick,Jane,Harry)

B(Tom,Jane,Dick,Harry)

B(Tom,Tom,Tom,Tom)

C(?1,?1,Tom,Tom)

C(?1,Dick,?2,Tom) D(?1,Dick,Tom,?2)

C(?2,?2,Jane,Jane) D(?2,?2,Tom,Tom)

C(?2,Dick,?4,Tom)

C(?2,Dick,Jane,Tom) B(?2,Dick,Tom,?4)

C(?2,Dick,Tom,Jane) B(?4,?4,Harry,Harry)

C(?4,?4,Tom,Tom)

C(?4,Dick,Harry,Tom) D(?4,Dick,Tom,Harry)

C(Dick,?1,?2,Tom)

C(Dick,?1,Tom,?2)

C(Dick,?2,?4,Tom)

C(Dick,?2,Jane,Tom)

C(Dick,?2,Tom,?4)

C(Dick,?2,Tom,Jane)

C(Dick,?4,Harry,Tom) D(Dick,?4,Tom,Harry)

C(Dick,Dick,?2,?2) B(Dick,Dick,?4,?4)

C(Dick,Dick,Harry,Harry) B(Dick,Dick,Jane,Jane)

C(Dick,Dick,Tom,Tom) B(Dick,Jane,Harry,Tom) D(Dick,Jane,Tom,Harry)

C(Dick,Tom,?4,Harry)

C(Dick,Tom,Harry,?4)

C(Dick,Tom,Harry,Jane)

C(Dick,Tom,Jane,Harry) B(Jane,Dick,Harry,Tom) D(Jane,Dick,Tom,Harry)

C(Jane,Jane,Harry,Harry)

C(Jane,Jane,Tom,Tom)

C(Tom,Dick,?4,Harry)

C(Tom,Dick,Harry,?4) D(Tom,Dick,Harry,Jane)

C(Tom,Dick,Jane,Harry) D(Tom,Tom,?4,?4) D(Tom,Tom,Harry,Harry)

C(Tom,Tom,Jane,Jane) A(?1,?2,Tom,Dick)

D(?1,Dick,?2,Tom)

D(?1,Dick,Tom,?2) A(?1,Tom,?2,Dick)

D(?2,?4,Tom,Dick) A(?2,Dick,?4,Tom)

D(?2,Dick,Jane,Tom) A(?2,Dick,Tom,?4)

D(?2,Dick,Tom,Jane) A(?2,Jane,Tom,Dick)

D(?2,Tom,?4,Dick)

D(?2,Tom,Jane,Dick)

D(?4,Dick,Harry,Tom)

D(?4,Dick,Tom,Harry)

D(?4,Harry,Tom,Dick)

D(?4,Tom,Harry,Dick) A(Dick,?1,?2,Tom)

D(Dick,?1,Tom,?2) B(Dick,?2,?4,Tom)

D(Dick,?2,Jane,Tom)

D(Dick,?2,Tom,?1) A(Dick,?2,Tom,?4)

D(Dick,?2,Tom,Jane)

D(Dick,?4,Harry,Tom) B(Dick,?4,Harry,Tom)

D(Dick,?4,Tom,?2) B(Dick,?4,Tom,Harry)

D(Dick,Harry,?4,Tom) B(Dick,Harry,Jane,Tom) A(Dick,Harry,Tom,?4)

D(Dick,Harry,Tom,Jane)

D(Dick,Jane,Harry,Tom)

D(Dick,Jane,Harry,Tom) B(Dick,Jane,Tom,?2)

D(Dick,Jane,Tom,Harry)

D(Dick,Tom,?2,?1) A(Dick,Tom,?4,?2)

D(Dick,Tom,?4,Harry)

D(Dick,Tom,Harry,?4)

D(Dick,Tom,Harry,?4) B(Dick,Tom,Harry,Jane)

D(Dick,Tom,Harry,Jane) B(Dick,Tom,Jane,?2)

D(Dick,Tom,Jane,Harry)

D(Jane,Dick,Harry,Tom) A(Jane,Dick,Tom,Harry)

D(Jane,Harry,Tom,Dick)

D(Jane,Tom,Harry,Dick)

D(Tom,?4,Harry,Dick)

D(Tom,Dick,?4,Harry)

D(Tom,Dick,Harry,?4)

D(Tom,Dick,Harry,Jane)

D(Tom,Dick,Jane,Harry)

D(Tom,Harry,?4,Dick) A(Tom,Harry,Jane,Dick)

D(Tom,Jane,Harry,Dick)

PROOFS ON THE MARCUS SET:

KB:

-isHuman(?1) OR mortal(?1)

-mortal(?4) OR -yearsSinceBirth(?4,?2) OR -greaterThan(?2,150) OR isDead(?4,?3)

-isPompeian(?5) OR diedBefore(?5,80)

hasErupted(volcano,79)

greaterThan(1969,150)

isHuman(Marcus)

isPompeian(Marcus)

yearsSinceBirth(Marcus,1969)

TRYING TO PROVE THAT : isDead(Marcus,1969)

-isDead(Marcus,1969)

\

\ / -mortal(?4) OR -yearsSinceBirth(?4,?2) OR -greaterThan(?2,150) OR isDead(?4,?3)

\ / { ?4/Marcus ?3/1969 }

\/

-mortal(Marcus) OR -yearsSinceBirth(Marcus,?6) OR -greaterThan(?6,150)

\

\ / -isHuman(?1) OR mortal(?1)

\ / { ?1/Marcus }

\/

-yearsSinceBirth(Marcus,?7) OR -greaterThan(?7,150) OR -isHuman(Marcus)

\

\ / greaterThan(1969,150)

\ / { ?7/1969 }

\/

-yearsSinceBirth(Marcus,1969) OR -isHuman(Marcus)

\

\ / isHuman(Marcus)

\ / { }

\/

-yearsSinceBirth(Marcus,1969)

\

\ / yearsSinceBirth(Marcus,1969)

\ / { }

```
     \/
     NIL

THEREFORE :    isDead(Marcus,1969)


TRYING TO PROVE THAT :    -isDead(Marcus,1969)
COULD NOT PROVE THAT :    -isDead(Marcus,1969)
```

PROOFS ON THE BUTTERFLY SET:

KB:

```
    -isInsect(?1) OR -hasTwoWings(?1) OR -wasCaterpillar(?1) OR isButterfly(?1)
    -isOrange(?2) OR -migrates(?2) OR -isButterfly(?2) OR isMonarch(?2)
    isInsect(?3) OR -isButterfly(?3)
    hasTwoWings(?4) OR -isButterfly(?4)
    wasCaterpillar(?5) OR -isButterfly(?5)
    isButterfly(?6) OR -isMonarch(?6)
    isOrange(?7) OR -isMonarch(?7)
    migrates(?8) OR -isMonarch(?8)
    isInsect(Bubba)
    hasTwoWings(Bubba)
    wasCaterpillar(Bubba)
    isOrange(Bubba)
    migrates(Bubba)
    isInsect(Chuck)
    hasTwoWings(Chuck)
    wasCaterpillar(Chuck)
    -isOrange(Chuck)
    migrates(Chuck)
```

TRYING TO PROVE THAT : isMonarch(Bubba)

```
-isMonarch(Bubba)
  \
   \    /  -isOrange(?2) OR -migrates(?2) OR -isButterfly(?2) OR isMonarch(?2)
    \  /   { ?2/Bubba }
     \/
-isOrange(Bubba) OR -migrates(Bubba) OR -isButterfly(Bubba)
  \
   \    /  -isInsect(?1) OR -hasTwoWings(?1) OR -wasCaterpillar(?1) OR
isButterfly(?1)
    \  /   { ?1/Bubba }
     \/
-isOrange(Bubba) OR -migrates(Bubba) OR -isInsect(Bubba) OR -hasTwoWings(Bubba) OR
-wasCaterpillar(Bubba)
  \
   \    /  isInsect(Bubba)
```

```
    \  /   { }
     \/
-isOrange(Bubba) OR -migrates(Bubba) OR -hasTwoWings(Bubba) OR
-wasCaterpillar(Bubba)
  \
   \    /  hasTwoWings(Bubba)
    \  /   { }
     \/
-isOrange(Bubba) OR -migrates(Bubba) OR -wasCaterpillar(Bubba)
  \
   \    /  wasCaterpillar(Bubba)
    \  /   { }
     \/
-isOrange(Bubba) OR -migrates(Bubba)
  \
   \    /  isOrange(Bubba)
    \  /   { }
     \/
-migrates(Bubba)
  \
   \    /  migrates(Bubba)
    \  /   { }
     \/
     NIL

THEREFORE :    isMonarch(Bubba)



TRYING TO PROVE THAT :    -isMonarch(Chuck)


isMonarch(Chuck)
  \
   \    /  isOrange(?7) OR -isMonarch(?7)
    \  /   { ?7/Chuck }
     \/
isOrange(Chuck)
  \
   \    /  -isOrange(Chuck)
    \  /   { }
```

```
     \/
     NIL

THEREFORE :    -isMonarch(Chuck)


TRYING TO PROVE THAT :    isMonarch(Chuck)
COULD NOT PROVE THAT :    isMonarch(Chuck)
```

PROOFS ON THE GENERIC SET:

KB:

A(?1,?2,Tom,Dick) OR B(?1,Dick,?2,Tom)

A(?3,Dick,Tom,?4) OR C(?3,Tom,?4,Dick)

A(?6,?8,Dick,Tom)

A(?10,?12,Tom,Dick) OR B(?10,Dick,?12,Tom)

A(?14,Dick,Jane,Tom)

A(?16,Dick,Tom,?18)

A(?20,Dick,Tom,Jane)

A(?22,Jane,Dick,Tom)

A(?24,Jane,Tom,Dick) OR B(?24,Tom,?23,Dick)

A(?26,Tom,?27,?28) OR B(?26,Tom,?28,Dick)

A(?30,Tom,?32,Dick)

A(?34,Tom,Dick,Jane) OR C(?34,Tom,Jane,Dick)

A(?38,Dick,Harry,Tom)

A(?42,Dick,Tom,Harry)

A(?46,Harry,Dick,Tom) OR B(?46,Harry,Tom,Dick)

A(?50,Tom,?48,Dick)

A(?54,Tom,Harry,Dick)

A(Dick,?55,?56,Tom)

A(Dick,?57,Tom,?58) OR C(Dick,?58,?57,Tom)

A(Dick,?60,?62,Tom)

A(Dick,?64,Jane,Tom)

A(Dick,?66,Tom,?65)

A(Dick,?68,Tom,?70) OR B(Dick,?68,Tom,Jane)

A(Dick,?74,?72,Tom)

A(Dick,?78,Harry,Tom)

A(Dick,?82,Tom,?80)

A(Dick,?86,Tom,Harry)

A(Dick,Harry,?90,Tom) OR C(Dick,Harry,Jane,Tom)

A(Dick,Harry,Tom,?94)

A(Dick,Harry,Tom,Jane)

A(Dick,Jane,?96,Tom)

A(Dick,Jane,Harry,Tom)

A(Dick,Jane,Tom,?98) OR B(Dick,Jane,Tom,Harry)

A(Dick,Tom,?99,?100)

A(Dick,Tom,?102,?101)

A(Dick,Tom,?104,?106)

A(Dick,Tom,?108,Jane)
A(Dick,Tom,?112,?110)
A(Dick,Tom,?116,Harry) OR C(Dick,Tom,Harry,?116)
A(Dick,Tom,Harry,Jane)
A(Dick,Tom,Jane,?118)
A(Dick,Tom,Jane,Harry)
A(Harry,?122,Dick,Tom)
A(Harry,?126,Tom,Dick)
A(Harry,Jane,Dick,Tom)
A(Harry,Jane,Tom,Dick) OR B(Harry,Tom,?130,Dick)
A(Harry,Tom,Dick,?134)
A(Harry,Tom,Dick,Jane) OR C(Harry,Tom,Jane,Dick)
A(Jane,Dick,Harry,Tom) OR C(Jane,Dick,Tom,Harry)
A(Jane,Harry,Dick,Tom)
A(Jane,Harry,Tom,Dick)
A(Jane,Tom,?136,Dick)
A(Jane,Tom,Harry,Dick) OR B(Tom,?138,?137,Dick)
A(Tom,?140,Dick,?139)
A(Tom,?144,?142,Dick)
A(Tom,?148,Dick,?146) OR C(Tom,?148,Dick,Harry)
A(Tom,?152,Harry,Dick)
A(Tom,Dick,?156,Harry)
A(Tom,Dick,Harry,?160)
A(Tom,Dick,Harry,Jane)
A(Tom,Dick,Jane,Harry)
A(Tom,Harry,?164,Dick)
A(Tom,Harry,Dick,?168)
A(Tom,Harry,Dick,Jane) OR B(Tom,Harry,Jane,Dick)
A(Tom,Jane,?170,Dick)
A(Tom,Jane,Dick,?172)
A(Tom,Jane,Dick,Harry)
A(Tom,Jane,Harry,Dick) OR D(?173,Dick,?174,Tom)
B(?175,Dick,Dick,?176) OR C(?175,Dick,Dick,Tom)
B(?177,Dick,Tom,?178)
B(?180,?179,Dick,Tom)
B(?182,Dick,?181,Tom) OR D(?182,Dick,?184,Tom)
B(?186,Dick,Dick,?188)
B(?190,Dick,Dick,Jane)
B(?192,Dick,Dick,Tom)

B(?194,Dick,Jane,Tom) OR C(?194,Dick,Tom,?196)
B(?198,Dick,Tom,Jane)
-B(?200,Tom,?202,Dick)
B(?206,?204,Dick,Tom)
B(?210,Dick,?208,Tom) OR D(?210,Dick,Dick,Harry)
B(?214,Dick,Dick,Tom)
B(?218,Dick,Harry,Tom)
B(?222,Dick,Tom,Harry)
B(?226,Tom,Dick,Harry)
B(Dick,?227,?227,Tom) OR C(Dick,?227,?228,Tom)
B(Dick,?230,?230,Jane)
B(Dick,?232,?232,Tom) OR D(Dick,?232,?234,Tom)
B(Dick,?236,Jane,Tom) OR D(Dick,?236,Tom,Jane)
B(Dick,?240,?240,Harry)
B(Dick,?244,?244,Tom) OR C(Dick,?244,Harry,Tom)
B(Dick,?248,Tom,Harry)
B(Dick,Jane,Harry,Tom)
B(Dick,Jane,Jane,Harry)
B(Dick,Jane,Jane,Tom) OR D(Dick,Jane,Tom,Harry)
B(Dick,Tom,?252,Harry) OR D(Dick,Tom,Jane,Harry)
B(Dick,Tom,Tom,Harry)
B(Dick,Tom,Tom,Jane)
B(Harry,?256,Dick,Tom) OR C(Harry,Dick,?256,Tom)
B(Harry,Dick,Jane,Tom)
B(Harry,Dick,Tom,?260) OR D(Harry,Dick,Tom,Jane)
B(Harry,Harry,Harry,Harry) OR D(Harry,Jane,Dick,Tom)
B(Harry,Tom,Dick,Jane) OR D(Jane,?262,Dick,Tom)
B(Jane,Dick,?264,Tom) OR D(Jane,Dick,Dick,Harry)
B(Jane,Dick,Dick,Tom)
B(Jane,Dick,Harry,Tom) OR C(Jane,Dick,Tom,Harry)
B(Jane,Jane,Jane,Jane)
B(Jane,Tom,Dick,Harry)
B(Tom,?266,Dick,Jane)
B(Tom,?270,Dick,Harry)
B(Tom,Dick,?271,?272)
B(Tom,Dick,?274,?276)
B(Tom,Dick,?278,Jane)
B(Tom,Dick,?282,Harry) OR C(Tom,Dick,Dick,?282)
B(Tom,Dick,Dick,Harry)

B(Tom,Dick,Dick,Jane) OR D(Tom,Dick,Harry,?286) OR C(Tom,Dick,Harry,Jane)

B(Tom,Dick,Jane,Harry)

B(Tom,Jane,Dick,Harry)

B(Tom,Tom,Tom,Tom)

C(?287,?287,Tom,Tom)

C(?288,Dick,?289,Tom) OR D(?288,Dick,Tom,?289)

C(?291,?291,Jane,Jane) OR D(?291,?291,Tom,Tom)

C(?293,Dick,?295,Tom)

C(?297,Dick,Jane,Tom) OR B(?297,Dick,Tom,?299)

C(?301,Dick,Tom,Jane) OR B(?303,?303,Harry,Harry)

C(?307,?307,Tom,Tom)

C(?311,Dick,Harry,Tom) OR D(?311,Dick,Tom,Harry)

C(Dick,?312,?313,Tom)

C(Dick,?314,Tom,?315)

C(Dick,?317,?319,Tom)

C(Dick,?321,Jane,Tom)

C(Dick,?323,Tom,?325)

C(Dick,?327,Tom,Jane)

C(Dick,?331,Harry,Tom) OR D(Dick,?331,Tom,Harry)

C(Dick,Dick,?333,?333) OR B(Dick,Dick,?335,?335)

C(Dick,Dick,Harry,Harry) OR B(Dick,Dick,Jane,Jane)

C(Dick,Dick,Tom,Tom) OR B(Dick,Jane,Harry,Tom) OR D(Dick,Jane,Tom,Harry)

C(Dick,Tom,?339,Harry)

C(Dick,Tom,Harry,?343)

C(Dick,Tom,Harry,Jane)

C(Dick,Tom,Jane,Harry) OR B(Jane,Dick,Harry,Tom) OR D(Jane,Dick,Tom,Harry)

C(Jane,Jane,Harry,Harry)

C(Jane,Jane,Tom,Tom)

C(Tom,Dick,?347,Harry)

C(Tom,Dick,Harry,?351) OR D(Tom,Dick,Harry,Jane)

C(Tom,Dick,Jane,Harry) OR D(Tom,Tom,?355,?355) OR D(Tom,Tom,Harry,Harry)

C(Tom,Tom,Jane,Jane) OR A(?356,?357,Tom,Dick)

D(?358,Dick,?359,Tom)

D(?360,Dick,Tom,?361) OR A(?360,Tom,?361,Dick)

D(?363,?365,Tom,Dick) OR A(?363,Dick,?365,Tom)

D(?367,Dick,Jane,Tom) OR A(?367,Dick,Tom,?369)

D(?371,Dick,Tom,Jane) OR A(?371,Jane,Tom,Dick)

D(?373,Tom,?375,Dick)

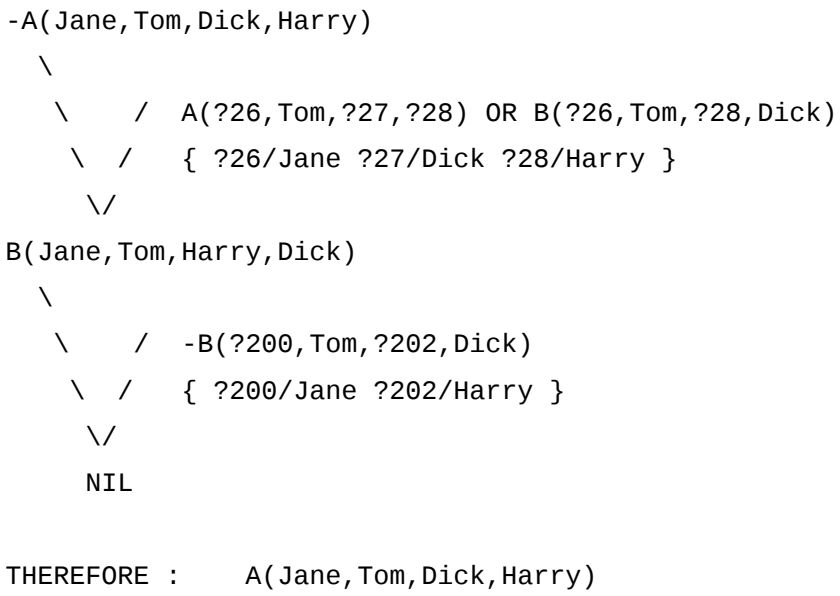
D(?377,Tom,Jane,Dick)

```
    D(?381,Dick,Harry,Tom)
    D(?385,Dick,Tom,Harry)
    D(?389,Harry,Tom,Dick)
    D(?393,Tom,Harry,Dick) OR A(Dick,?390,?391,Tom)
    D(Dick,?394,Tom,?395) OR B(Dick,?395,?397,Tom)
    D(Dick,?399,Jane,Tom)
    D(Dick,?401,Tom,?400) OR A(Dick,?401,Tom,?403)
    D(Dick,?405,Tom,Jane)
    D(Dick,?409,Harry,Tom) OR B(Dick,?409,Harry,Tom)
    D(Dick,?413,Tom,?411) OR B(Dick,?413,Tom,Harry)
    D(Dick,Harry,?417,Tom) OR B(Dick,Harry,Jane,Tom) OR A(Dick,Harry,Tom,?417)
    D(Dick,Harry,Tom,Jane)
    D(Dick,Jane,Harry,Tom)
    D(Dick,Jane,Harry,Tom) OR B(Dick,Jane,Tom,?419)
    D(Dick,Jane,Tom,Harry)
    D(Dick,Tom,?421,?420) OR A(Dick,Tom,?423,?421)
    D(Dick,Tom,?427,Harry)
    D(Dick,Tom,Harry,?431)
    D(Dick,Tom,Harry,?435) OR B(Dick,Tom,Harry,Jane)
    D(Dick,Tom,Harry,Jane) OR B(Dick,Tom,Jane,?437)
    D(Dick,Tom,Jane,Harry)
    D(Jane,Dick,Harry,Tom) OR A(Jane,Dick,Tom,Harry)
    D(Jane,Harry,Tom,Dick)
    D(Jane,Tom,Harry,Dick)
    D(Tom,?441,Harry,Dick)
    D(Tom,Dick,?445,Harry)
    D(Tom,Dick,Harry,?449)
    D(Tom,Dick,Harry,Jane)
    D(Tom,Dick,Jane,Harry)
    D(Tom,Harry,?453,Dick) OR A(Tom,Harry,Jane,Dick)
    D(Tom,Jane,Harry,Dick)

TRYING TO PROVE THAT :    Z(Bubba)

COULD NOT PROVE THAT :    Z(Bubba)


TRYING TO PROVE THAT :    A(Jane,Tom,Dick,Harry)
```



THEREFORE : A(Jane,Tom,Dick,Harry)