lib/basic/prolog.ath

```
#load "list-of"

module Prolog {

define print-info-flag := (cell true)

define (print-info) := (set! print-info-flag true)

define (dont-print-info) := (set! print-info-flag false)

define (mprint str) :=
  check {(ref print-info-flag) => (print str)
       | else => ()}

# Embedding of SWI Prolog into Athena. First draft
# checked in on October 01, 2011, by K.A.

# Athena-SWI counterparts of some standard Prolog predicates:
# is, ==, \==, =:=, and =\=

declare is, ==, \==, arith-eq, arith-uneq: (T) [T T] -> Boolean

# Cut, fail, call, Prolog write, bagof, setof, and findall:

declare cut, fail: Boolean

declare write: [Ide] -> Boolean

declare call: [Boolean] -> Boolean

declare bagof, setof, findall: (T1,T2,T3) [T1 T2 T3] -> Boolean

define term-transformers := (cell [])

define (add-transformer T) := (set! term-transformers (add T (ref term-transformers)))

define (make-safe T) :=
  lambda (t)
    try { (T t) | t }

define (apply-transformer T) :=
  let {T' := (make-safe T)}
  lambda (sub)
    match sub {
      (some-sub sub) =>
        (map lambda (v)
               [v --> (T' (sub v))]
             (supp sub))
    | (some-list L) => (map lambda (triple)
                              match triple {
                                [(some-var v) --> (some-term t)] => [v --> (T' t)]
                              | _ => triple
                              }
                            L)
    }

define (transform-sub x) :=
  match x {
    (some-sub sub) => letrec {loop := lambda (transformers res)
                                        match transformers {
                                          [] => res
                                        | (list-of T more) => (loop more ((apply-transformer T) res))
                                        }
                             }
                        (loop (ref term-transformers) sub)
  | (some-list _) => (map transform-sub x)
  | _ => x
  }

declare goal1: (T1) [T1] -> Boolean
declare goal2: (T1, T2) [T1 T2] -> Boolean
declare goal3: (T1, T2, T3) [T1 T2 T3] -> Boolean
declare goal4: (T1, T2, T3, T4) [T1 T2 T3 T4] -> Boolean
declare goal5: (T1, T2, T3, T4, T5) [T1 T2 T3 T4 T5] -> Boolean
declare goal6: (T1, T2, T3, T4, T5, T6) [T1 T2 T3 T4 T5 T6] -> Boolean
declare goal7: (T1, T2, T3, T4, T5, T6, T7) [T1 T2 T3 T4 T5 T6 T7] -> Boolean
declare goal8: (T1, T2, T3, T4, T5, T6, T7, T8) [T1 T2 T3 T4 T5 T6 T7 T8] -> Boolean
declare goal9: (T1, T2, T3, T4, T5, T6, T7, T8, T9) [T1 T2 T3 T4 T5 T6 T7 T8 T9] -> Boolean
declare goal10: (T1, T2, T3, T4, T5, T6, T7, T8, T9, T10) [T1 T2 T3 T4 T5 T6 T7 T8 T9 T10] -> Boolean

define goal-vec := (make-vector 10 ())

(vector-set! goal-vec 0 goal1)
(vector-set! goal-vec 1 goal2)
(vector-set! goal-vec 2 goal3)
(vector-set! goal-vec 3 goal4)
(vector-set! goal-vec 4 goal5)
(vector-set! goal-vec 5 goal6)
(vector-set! goal-vec 6 goal7)
(vector-set! goal-vec 7 goal8)
(vector-set! goal-vec 8 goal9)
(vector-set! goal-vec 9 goal10)

(define (get-goal-predicate i)
  (vector-sub goal-vec (minus i 1)))

# Some hash tables necessary for getting around Prolog's
# syntactic idiosyncracies:

define [var-table var-table' constant-table constant-table' fsym-table fsym-table'] :=
       [(table 97) (table 97) (table 97) (table 97) (table 97) (table 97)];

define [var-counter constant-counter fsym-counter] :=
       [(cell 0) (cell 0) (cell 0)];

define clear-memory :=
  lambda ()
    (seq (table-clear var-table)
        (table-clear var-table')
        (table-clear constant-table)
        (table-clear constant-table')
        (table-clear fsym-table)
        (table-clear fsym-table')
        (set! var-counter 0)
        (set! constant-counter 0)
        (set! fsym-counter 0));

define make-fresh-var :=
  lambda ()
    ("V" joined-with (val->string inc var-counter))

(define (make-fresh-constant)
  (join "a" (val->string (inc constant-counter))))

(define (make-fresh-fun-sym)
  (join "h" (val->string (inc fsym-counter))))

(define (prolog-legal? str)
  (for-each str (lambda (c) (|| (alpha-numeric-char? c) (equal? c `_)))))

(define (make-prolog-constant c)
  (check ((integer-numeral? c) (check ((greater-or-equal? c 0) (val->string c))
                                      (else (join "-" (val->string (abs c))))))
         ((equal? c cut) "!")
         ((equal? c fail) "fail")
         ((equal? c true) "true")
         ((equal? c false) "fail")
         (else (let ((str (symbol->string c)))
                 (check ((prolog-legal? str) (add `c str))
                        (else (try (add `k (table-lookup constant-table str))
                                   (let ((str' (make-fresh-constant))
                                         (_ (table-add constant-table [str --> str']))
                                         (_ (table-add constant-table' [str' --> c])))
                                    (add `k str')))))))))

(define (numeric-op? f)
  (member? f [(string->symbol "+")
              (string->symbol "-")
              (string->symbol "*")
              (string->symbol "/")
              (string->symbol "<")
              (string->symbol ">")]))

(define (make-prolog-functor f)
  (check ((numeric-op? f) (symbol->string f))
         ((equal? f (string->symbol "<=")) "=<")
         ((equal? f (string->symbol ">=")) ">=")
         ((equal? f =) (symbol->string f))
         ((equal? f is) "is")
         ((equal? f ==) "==")
         ((equal? f \==) "\\==")
         ((equal? f arith-eq) "=:=")
         ((equal? f arith-uneq) "=\\=")
         ((equal? f call) "call")
         ((equal? f bagof) "bagof")
         ((equal? f setof) "setof")
         ((equal? f findall) "findall")
         (else (let ((str (symbol->string f)))
                (check ((prolog-legal? str) (add `f str))
                       (else (try (add `g (table-lookup fsym-table str))
                                  (let ((str' (make-fresh-fun-sym))
                                        (_ (table-add fsym-table [str --> str']))
                                        (_ (table-add fsym-table' [str' --> f])))
                                    (add `g str')))))))))


(define (make-prolog-term t)
  (match t
    ((some-var v) (let ((str (var->string v)))
                    (check ((prolog-legal? str) (add `X str))
                           (else (try (add `Y (table-lookup var-table str))
                                      (let ((str' (make-fresh-var))
                                            (_ (table-add var-table [str --> str']))
                                            (_ (table-add var-table' [str' --> v])))
                                         (add `Y str')))))))
    ((write s) (join "write('" (join "O" (id->string s) "\\n") "')"))
    (((some-symbol f) (some-list args))
           (match args
             ([] (make-prolog-constant f))
             (_ (join (make-prolog-functor f)
                      "("
                      (separate (map make-prolog-term args) ",")
                      ")"))))))

(define (make-prolog-prop p)
  (match p
    ((some-atom _) (make-prolog-term p))
    ((not (some-sent q)) (join "not(" (make-prolog-prop q) ")"))
    ((and (some-list args)) (separate (map make-prolog-prop args) ", "))
    ((or (some-list args)) (join "(" (separate (map make-prolog-prop args) "; ") ")"))
    ((if true consequent) (make-prolog-prop consequent))
    ((if antecedent consequent) (join (make-prolog-prop consequent)
                                      " :- "
                                      (make-prolog-prop antecedent)))
    ((forall (list-of _ _) (if (exists (list-of _ _) cond) (some-atom body)))
        (make-prolog-prop (if cond body)))
    ((forall (list-of _ _) (iff (some-atom body) (exists (list-of _ _) cond)))
        (make-prolog-prop (if cond body)))
    ((forall (list-of _ _) body) (make-prolog-prop body))
    ((iff (as left (some-atom _)) right) (make-prolog-prop (if right left)))
    ((iff left (as right (some-atom _))) (make-prolog-prop (if left right)))
    (_ "")))

(define [bar comma lparen rparen lbrack rbrack blank colon scolon quot-mark]
        [" | " "," "(" ")" "[" "]" " " ":" ";" "\""])

(define [c-comma c-lparen c-rparen c-blank c-newline] [`, `( `) `\blank `\n])

(define (white-space? c)
   (member? c [c-blank c-newline]))

(define (variable? x)
  (match x
    ((list-of (some-char c) _)  (|| (upper-case-alpha-char? c) (equal? c `_)))
    (_ false)))

(define (get-string str res)
  (match str
    ([] [(rev res) []])
    ((list-of (some-char c) rest)
      (check ((member? c [c-blank c-newline c-lparen c-rparen c-comma]) [(rev res) str])
             (else (get-string rest (add c res)))))))

(define (get-symbol str)
  (check ((member? str ["+" "-" "*" "/" "="]) (string->symbol str))
         ((all-digits? str) (string->symbol str))
         ((equal? str "[]") nil)
         ((equal? str "'.'") ::)
         (else (match str
                 ((list-of `c rest) (string->symbol rest))
                 ((list-of `k rest) (table-lookup constant-table' rest))
                 ((list-of `f rest) (string->symbol rest))
                 ((list-of `g rest) (table-lookup fsym-table' rest))))))

(define (fresh-variable? str)
  (equal? str "_"))

(define [parse-term parse-terms]
  (letrec ((get-term (lambda (str)
                       (match (get-string str [])
                         ([root (list-of `( rest)]
                             (match (get-terms rest [])
                                ([args (list-of `) rest')]
                                  (let ((fsym (get-symbol root))
                                        (res-term (make-term fsym args)))
                                    [res-term rest']))))
                         ([root rest] (let ((_ (mprint (join "\nroot: " (val->string root)))))
                                        (check ((fresh-variable? root) (let ((_ ())) [(fresh-var) rest]))
                                               ((variable? root) (match root
                                                                   ((list-of `Y more) [(table-lookup var-table' more)
                                                                   (_  (check
                                                                         ((null? (tail root))
                                                                           [(string->var root) rest])
                                                                         (else [(string->var (tail root))
                                                                                rest])))))
                                               (else (match root
                                                       ((list-of `- more) (let (([t more'] (get-term (join more rest))
                                                                                    [(make-term (string->symbol "-") [t]) more
                                                       (_ [(make-term (get-symbol root) []) rest])))))))))
           (get-terms (lambda (str terms)
                         (match (get-term str)
                           ([term (list-of `, rest)] (get-terms rest (add term terms)))
                           ([term rest] [(rev (add term terms)) rest])))))
       [get-term (lambda (str) (get-terms str []))]))


(define (get-line str)
  (letrec ((loop (lambda (str chars)
                    (match str
                      ([] [(rev chars) []])
                      ((list-of `\n rest) [(rev (add `\n chars)) rest])
                      ((list-of (some-char c) rest)  (loop rest (add c chars)))))))
    (loop str [])))


(define (process-output-line str v)
 (let ((_ (mprint (join "\nProcessing this line: " str "\n"))))
  (match str
    ((list-of `X more) (let (([var-name rest] (get-string more []))
                             (term (first (parse-term (skip-until rest printable?))))
                             (equality (= (string->var var-name) term))
                             (equality (= v term)))
                         [(lhs equality) (rhs equality)]))
    ((list-of `Y more) (let (([var-name rest] (get-string more []))
                             (term (first (parse-term (skip-until rest printable?))))
                             (equality (= (table-lookup var-table' var-name) term))
                             (equality (= v term)))
                         [(lhs equality) (rhs equality)]))
## Output line:
    ((list-of `O more) (seq (print more) ()))
    ((split "yes" _) true)
    ((split "no" _) false))))


(define (process-output data vars)
  (letrec ((loop (lambda (data results vars)
                   (match [data vars]
                     ([[] []] [true (rev results)])
                     ([[] _] [false (rev results)])
                     ([_ (list-of v more-vars)]
                       (match (get-line data)
                          ([line rest] (match (process-output-line line v)
                                         ([l r] (loop rest (add [l r] results) more-vars))
                                         (true (loop rest results more-vars))
                                         (false [false []])
                                         (() (loop rest results vars)))))))
                     ([_ []]
                       (match (get-line data)
                          ([line rest] (match (process-output-line line ())
                                         ([l r] (loop rest (add [l r] results) []))
                                         (true (loop rest results []))
                                         (false [false []])
                                         (() (loop rest results [])))))))))))
     (match (loop data [] vars)
       ([(some-term b) pairs] [b (make-sub pairs)]))))


define solve-with-time-limit-aux :=
lambda (program query-list time-limit)
  (let ((start-time (time))
        ([input-file output-file error-file] ["a.pl" "o.pl" "e.pl"])
        (_ (delete-files [input-file output-file]))
        (_ (clear-memory))
        (prolog-program (separate (map (lambda (p) (join (make-prolog-prop p) ".\n")) program) ""))
        (_ (seq (mprint "\nGiven program:\n") (mprint prolog-program) (mprint "\n")))
        (_ (write-file input-file "\nuse_module(library(time)).\n"))
        (_ (write-file input-file prolog-program))
        (_ (write-file input-file "\nverbose_eval(Predicate) :- call(Predicate) -> write(yes) ; write(no)."))
        (goal-string' (join "("
                            (separate (map (lambda (t)
                                      (check ((ground? t) (join "verbose_eval(" (make-prolog-prop t) ")"))
                                             (else (make-prolog-prop t))))
                                    query-list) ",")
                           ")"))
        (goal-string (check ((|| true (greater? time-limit 0))
                               (join "call_with_time_limit(" (val->string time-limit) "," goal-string' ")"))
                            (else goal-string')))
        (vars (rev (vars* query-list)))
        (var-strings (map make-prolog-term vars))
        (make-write-var-string (lambda (var-string)
                                 (join "write('" var-string "'),write(' '),write_canonical(" var-string "), nl")))
        (write-var-strings (match (separate (map make-write-var-string var-strings) ",")
                             ([] [])
                             (str (join str ","))))
        (write-var-strings’ (match (separate (map make-write-var-string var-strings) ",")
                              ([] [])
                              (str (join "," str))))
        (_ (mprint (join "\nGoal string: " (val->string goal-string))))
        (all-ground (for-each query-list ground?))
        #(param-file "prologparams.txt")
        #(param-string (join " -f " input-file " -g \"" goal-string write-var-strings’ "\" -t \"halt\" -q > " output-f
        #(_ (write-file param-file param-string))
        #(command (join "swipl  " param-file))
        (command (join "swipl -f " input-file " -g \"" goal-string write-var-strings’ "\" -t \"halt\" -q > " output-fi
        (_ (mprint (join "\nTIME UP TILL COMMAND EXECUTION: " (val->string (minus (time) start-time)))))
        (_ (mprint (join "\nCommand: " command)))
        (_ (exec-command command))
        (output-data (read-file output-file))
        (_ (mprint (join "\nOutput: " (val->string output-data)))))
     (match output-data
       ([] (check ((null? query-list) [true empty-sub])
                  (else [false empty-sub])))
       (str (process-output str vars))))

(define (solve-with-time-limit query-list program time-limit)
  (match query-list
   ((some-list _) (solve-with-time-limit-aux program query-list time-limit))
   (_ (solve-with-time-limit-aux program [query-list] time-limit))))

(define [LB RB COMMA BACKSLASH] [‘[ ‘] ‘, ‘/])

(define (accum-until stream pred)
  (letrec ((loop (lambda (stream res)
                   (match stream
                     ([] [(rev res) []])
                     ((list-of (some-char c) more)
                        (check ((pred c) [(rev res) stream])
                               (else (loop more (add c res)))))))))
     (loop stream [])))

(define (get-list-content stream)
   (accum-until stream (lambda (c) (equal? c RB))))

(define (get-one-list stream)
  (match (skip-until stream printable?)
    ((list-of (val-of LB) more) (let (([str rest] (get-list-content more)))
                                  [(first (parse-terms str)) (tail rest)]))))

(define (process-find-all-output str vars)
  (letrec ((loop (lambda (stream results)
                    (match (skip-until stream printable?)
                      ([] (rev results))
                      ([(val-of RB)] (rev results))
                      ((list-of (val-of COMMA) more)
                         (loop more results))
                      (_ (match (get-one-list stream)
                           ([(some-list terms) rest’]
                              (let ((sub-content (map (lambda (var-term-pair)
                                                         (match var-term-pair
                                                           ([v t] (let ((equality (= v t)))
                                                                    [(lhs equality) (rhs equality)]))))
                                                       (zip vars terms))))
                                (loop rest’ (add sub-content results)))))))))
    (loop (tail str) [])))

(define (process-find-all-output’ str vars)
  (match (process-find-all-output str vars)
    ((some-list sub-lists)(map make-sub sub-lists))))

(define numeric-term-portray-def
  (join "\nis_numeric(X) :- functor(X,+,_), !."
        "\nis_numeric(X) :- functor(X,-,_), !."
        "\nis_numeric(X) :- functor(X,*,_), !."
        "\nis_numeric(X) :- functor(X,/,_), !."
        "\nis_numeric(X) :- functor(X,<,_), !."
        "\nis_numeric(X) :- functor(X,<=,_), !."
        "\nis_numeric(X) :- functor(X,>,_), !."
        "\nis_numeric(X) :- functor(X,>=,_), !."
        "\nportray(X) :- is_numeric(X), write_canonical(X).\n"))

private define solve-all-with-time-limit-aux :=
lambda (program query-list time-limit)
  (let (([input-file output-file error-file] ["a.pl" "o.pl" "e.pl"])
        (_ (delete-files [input-file output-file error-file]))
        (_ (clear-memory))
        (prolog-program (separate (map (lambda (p) (join (make-prolog-prop p) ".\n")) program) ""))
        (_ (mprint (join "\nGiven program:\n" (val->string prolog-program))))
        (_ (write-file input-file "\nuse_module(library(time)).\n"))
        (_ (write-file input-file numeric-term-portray-def))
        (_ (write-file input-file prolog-program))
        (vars (rev (vars* query-list)))
        (var-strings (map make-prolog-term vars))
        (var-string (join "[" (separate var-strings ",") "]"))
        (answer-var (add `X (var->string (fresh-var))))
        (goal-string (join "("
                            (separate (map make-prolog-term query-list)
                                      ",")
                            ")"))
        (total-goal (check ((|| true (greater? time-limit 0))
                              (join "call_with_time_limit(" (val->string time-limit)
                                     ",findall(" var-string "," goal-string "," answer-var ")), write_term(" answer-va
                           (else (join "findall(" var-string "," goal-string "," answer-var "), write_term(" answer-va
        (_ (mprint (join "\nGoal string: " (val->string total-goal))))
        (command (join "swipl -f " input-file " -g \"" total-goal "\" -t \"halt\" -q > " output-file " 2> " error-file
        (_ (mprint (join "\nCommand: " command)))
        (_ (exec-command command))
        (output-data (read-file output-file))
        (_ (mprint (join "\nOutput: " output-data)))
        (_ ()))
     (match output-data
       ([] [])
       (str (process-find-all-output' str vars))))

private define solve-N-with-time-limit-aux :=
lambda (prog goal N time-limit)
  (letrec ((loop (lambda (i subs)
                   (check ((less? N i) (rev subs))
                          (else (let ((sub-negations (lambda (sub)
                                                        (map (lambda (v) (not (= v (sub v)))) (supp sub))))
                                      (all-negations (flatten (map sub-negations subs)))
                                      (goal' (join goal all-negations)))
                                  (match (solve-with-time-limit-aux prog goal' time-limit)
                                    ([false _] (rev subs))
                                    ([true (some-sub sub)] (loop (plus i 1) (add sub subs)))))))))))
    (check ((less? N 1) [])
           (else (loop 1 [])))))

(define (solve-N-with-time-limit goal prog N time-limit)
  (match goal
     ((some-list _) (solve-N-with-time-limit-aux prog goal N time-limit))
     (_ (solve-N-with-time-limit-aux prog [goal] N time-limit))))

(define MAX-TIME-LIMIT 10000000)

define solve-aux :=
lambda (prog g)
  (match g
    ((some-list _) (solve-with-time-limit-aux prog g MAX-TIME-LIMIT))
    ((some-sentence _) (solve-with-time-limit-aux prog [g] MAX-TIME-LIMIT))
    (_ (error "Prolog.solve error: the goal must be either a sentence or a list of sentences.")))

(define (solve g prog)
  (transform-sub (solve-aux prog g)))

(define (solve-all-with-time-limit query-list program time-limit)
  (match query-list
    ((some-list _) (solve-all-with-time-limit-aux program query-list time-limit))
    (_ (solve-all-with-time-limit-aux program [query-list] time-limit))))

define solve-all-aux :=
lambda (prog g)
  (match g
    ((some-list _)   (solve-all-with-time-limit-aux prog g MAX-TIME-LIMIT))
    ((some-sentence _) (solve-all-with-time-limit-aux prog [g] MAX-TIME-LIMIT))
    (_ (error "Prolog.solve-all error: the goal must be either a sentence or a list of sentences.")))

(define (solve-all g prog)
  (transform-sub (solve-all-aux prog g)))

define solve-N-aux :=
lambda (prog goal N)
  (match goal
    ((some-list _)   (solve-N-with-time-limit-aux prog goal N MAX-TIME-LIMIT))
    ((some-sentence _) (solve-N-with-time-limit-aux prog [goal] N MAX-TIME-LIMIT))
    (_ (error "Prolog.solve-N error: the goal must be either a sentence or a list of sentences.")))

(define (solve-N goal prog N)
  (solve-N-aux prog goal N))

(define (make-new-clause g-pred goal-vars goals subs)
  (let ((negate-sub (lambda (sub)
                      (or (map (lambda (v)
                                 (not (= v (sub v))))
                               goal-vars))))
        (head (make-term g-pred goal-vars))
        (body (and (join goals (map negate-sub subs)))))
   (if body head)))

(define (solve-N-aux goals prog N lim)
  (let ((goals (match goals ((some-list _) goals) (_ [goals])))
        (goal-vars (vars* goals))
        (var-num (length goal-vars))
        (g-pred (get-goal-predicate var-num))
        )
    (letrec ((loop (lambda (i subs)
                     (check ((less? N i) (rev subs))
                            (else (let ((new-clause (make-new-clause g-pred goal-vars goals subs))
                                        (new-goal (make-term g-pred goal-vars))
                                        (new-prog (join prog [new-clause]))
                                        (res (check ((less? lim 0) (solve-aux new-prog new-goal))
                                                    (else (solve-with-time-limit-aux new-prog new-goal)))))
                                    (match res
                                      ([false _] (rev subs))
                                      ([_ (some-sub sub)] (loop (plus i 1) (add sub subs))))))))))
       (loop 1 []))))

(define (solve-N goals prog N)
  (transform-sub (solve-N-aux goals prog N (- 1))))

(define (solve-N-with-time-limit goals prog N lim)
  (transform-sub (solve-N-aux goals prog N lim)))

} # module Prolog

module Horn {

(define pred-table (table 100))
```

```
(define (make-fresh-pred-name f)
  (let ((symbol? (lambda (str)
                    (try (seq (string->symbol str) true) false)))
        (index (cell 0)))
    (letrec ((loop (lambda (prefix name)
                     (check ((symbol? name) (loop prefix (join prefix (val->string (inc index)))))
                            (else name)))))
      (let ((first-try (map downcase (join (map  (lambda (c) (check ((equal? c `.) `_) (else c))) (val->string f)) "_P
         (loop first-try first-try)))))

(define (boolean-symbol? f)
  (equal? (last (get-signature f)) "Boolean"))

(define (get-pred-version f)
 (let ((f (get-symbol f)))
  (check ((boolean-symbol? f) f)
         (else  (try (table-lookup pred-table f)
                     (let ((f-pred-name (make-fresh-pred-name f))
                           (sig (get-signature f))
                           (sort-string (separate sig " "))
                           (toks (tokenize-string sort-string [`’]))
                           (toks (filter toks (lambda (t) (equal? (first t) `T))))
                           (toks (dedup (map (lambda (t) (first (tokenize-string t [`\blank `( `)]))) toks)))
                           (sort-var-string (check ((null? toks) "") (else (join "(" (separate toks ", ") ")"))))
                           (sort-string’ (filter-out sort-string (lambda (c) (equal? c `’))))
                           (cmd (join "declare " f-pred-name ": " sort-var-string " [" sort-string’ "] -> Boolean"))
                           (_ (process-input-from-string cmd true))
                           (pf (string->symbol f-pred-name))
                           (_ (table-add pred-table [f --> pf])))
                       pf))))))


(define (term->horn-clause t)
  (match t
    ((|| (some-var _) ((some-symbol _) [])) [[] t])
    (((some-symbol f) (some-list args))
      (check ((constructor? f)
               (let (([arg-clauses arg-vars] (unzip (map term->horn-clause args)))
                      (arg-clauses (flatten (map join arg-clauses))))
                 [arg-clauses (make-term f arg-vars)]))
              ((boolean-symbol? f)
                  (let (([arg-clauses arg-vars] (unzip (map term->horn-clause args)))
                        (arg-clauses (flatten (map join arg-clauses))))
                     [(join arg-clauses [(make-term f arg-vars)]) ()]))
              (else (match args
                      ([]  [[] t])
                      (_ (let (([arg-clauses arg-vars] (unzip (map term->horn-clause args)))
                               (arg-clauses (flatten (map join arg-clauses)))
                               (out-var (fresh-var))
                               (last-clause (make-term (get-pred-version f) (join arg-vars [out-var]))))
                           [(join arg-clauses [last-clause]) out-var]))))))))

(define thc term->horn-clause)

(define (literal->hc t)
  (match t
    ((not (some-term t))
       (match (term->horn-clause t)
         ([(clauses as (list-of _ _)) ()] [(join (all-but-last clauses) [(not (last clauses))]) ()])
         ([clauses (some-term bool-term)] [(join clauses [(not bool-term)]) ()])))
    ((some-term _) (term->horn-clause t))))

(define (get-all-clauses bool-terms)
  (let (([clauses _] (unzip (map literal->hc bool-terms))))
    (flatten (map join clauses))))

(define (smart-and L)
  (match L
    ([(some-sent p)] p)
    ([] true)
    (_ (and L))))

(define (eqn->horn-clause-aux eqn)
  (match eqn
    ((forall (some-list _) (= (l as ((some-symbol f) (some-list args)))
                              (some-term r)))
      (check ((boolean-symbol? f)
                (match (term->horn-clause r)
                   ([clauses ()] (if (smart-and clauses) l))
                   ([clauses bool-term] (if (smart-and (join clauses [bool-term])) l))))
             (else (let ((fp (get-pred-version f)))
                     (match (term->horn-clause r)
                       ([clauses out] (if (smart-and clauses) (make-term fp (join args [out])))))))))
    ((forall (some-list _)
             (|| (if (ant as (|| (some-term guard) (guard as (not (some-term _))))) (body as (= (l as ((some-symbol f)
                 (if (ant as (and (some-list guards))) (body as (= (l as ((some-symbol f) (some-list args))) (some-ter
      (let ((guards (try [guard] guards))
            (conjuncts (get-conjuncts-recursive ant))
            (clauses (get-all-clauses conjuncts)))
        (match (eqn->horn-clause-aux body)
           ((if (some-sent ant) (some-sent con))
              (let ((clauses' (join clauses (get-conjuncts-recursive ant)))
                    (clauses' (filter-out clauses' (lambda (c) (equal? c true)))))
                (if (smart-and clauses') con))))))))

(define (eqn->horn-clauses eqn)
  (match eqn
    ((forall (list-of _ _) (if (exists (list-of _ _) (some-sent ant)) (some-atom body)))
      (eqn->horn-clauses (if ant body)))
    ((forall (list-of _ _) (iff (some-atom body) (exists (list-of _ _) cond)))
      (eqn->horn-clauses (if cond body)))
    ((forall (some-list _) (if (ant as (or (some-list guards))) con))
       (let ((D (get-disjuncts-recursive ant))
             (eqns (map (lambda (d) (if d con)) D))
             (clauses (map eqn->horn-clause-aux eqns)))
         clauses))
    (_ (try [(eqn->horn-clause-aux eqn)] []))))

(define (post-process clause)
  (match clause
    ((if true (some-sent body)) body)
    ((if (ant as (|| (some-term cond)
                     (and (some-list conds)))) body)
     (let ((conds (try [cond] conds))
           (body-vars (vars body)))
      (letrec ((loop (lambda (clauses idents non-idents)
                        (match clauses
                          ([] [(rev idents) (rev non-idents)])
                          ((list-of (c as (= (some-var x) _)) (some-list rest))
                              (check ((|| (member? x (vars* rest)) (negate (member? x body-vars)))
                                        (loop rest idents (add c non-idents)))
                                     (else (loop rest (add c idents) non-idents))))
                          ((list-of (some-sent c) (some-list rest))
                              (loop rest idents (add c non-idents)))))))
       (let (([identities non-idents] (loop (get-conjuncts-recursive ant) [] []))
             (bindings (map (lambda (i) (match i ((= (some-term l) (some-term r)) [l r]))) identities))
             (sub (make-sub bindings)))
          (match non-idents
            ([] (sub body))
            (_ (if (smart-and non-idents) (sub body))))))))
    (_ clause)))

(define pp post-process)

(define (ehc eqn) (map post-process (eqn->horn-clauses eqn)))

### make-horn-clauses is the official procedure for taking a (possibly conditional) equation
### and turning it into a list of horn clauses:
```

```
(define (make-horn-clauses L)
  (match L
    ((some-sent p) (ehc p))
    ((some-list _) (flatten (map ehc L)))))

(define mhc make-horn-clauses)

(define (test-sym f)
  (let ((eqns (map quant-body (defining-axioms f))))
    (flatten (map ehc eqns))))

(define (get-syms x)
  (match x
    ((some-term _) (get-term-syms x))
    ((some-sent _) (get-prop-syms x))
    ((some-list _) (flatten (map get-syms x)))))

(define (occurring-syms s)
 (try
  (let ((m (fsd s)))
    (match (m 'occurring-syms)
      ((some-list sym-names) (map string->symbol sym-names))
      (_ [])))
  []))

(define (guard-syms s)
 (try
  (let ((m (fsd s)))
    (match (m 'guard-syms)
      ((some-list sym-names) (map string->symbol sym-names))
      (_ [])))
  []))

(define (get-all-syms goal)
  (let ((syms0 (dedup (get-syms goal)))
        (syms0 (filter-out syms0 (lambda (s) (null? (defining-axioms s)))))
        (T (table 100))
        (_ (map-proc (lambda (s) (table-add T [s --> true])) syms0))
        (reachable-syms (lambda (s)
                          (let (([osyms gsyms] [(occurring-syms s) (guard-syms s)]))
                            (join osyms gsyms))))
        (reachable-syms* (lambda (syms)
                           (filter-out (dedup (flatten (map reachable-syms syms)))
                                       (lambda (s) (null? (defining-axioms s))))))
        (existing? (lambda (s) (try (table-lookup T s) false))))
    (letrec ((loop (lambda (syms)
                     (let ((syms' (filter-out (reachable-syms* syms) existing?))
                           (_ (map-proc (lambda (s) (table-add T [s --> true])) syms')))
                       (check ((null? syms') ())
                              (else (loop syms')))))))
        (let ((_ (loop syms0)))
          (dedup (map first (table->list T)))))))


(define (get-all-syms-sorted goal)
   (let ((L (get-all-syms goal)))
     (prim-sort L (lambda (s1 s2)
                    (try (let ((m (fsd s1)))
                           (match (m 'needed-by-syms)
                              ((some-list sym-names) (member? s2 (map string->symbol sym-names)))
                              (_ false)))
                         false)))))

(define (sorted-defining-axioms s)
  (prim-sort (defining-axioms s)
             (lambda (p1 p2) (less? (size p1) (size p2)))))

(define (get-all-horn-clauses goal)
  (let ((syms (get-all-syms-sorted goal))
        (all-clauses (flatten (map (lambda (s)
                                      (make-horn-clauses (sorted-defining-axioms s)))
                                     syms))))
     (dedup all-clauses)))

(define (test goal)
  (get-all-horn-clauses goal))

(define (make-goal p)
  (make-horn-clauses p))

(define (solve goals)
  (let ((program-clauses (get-all-horn-clauses goals))
        (goals (match goals ((some-list _) goals) (_ [goals])))
        (goal-clauses (get-all-clauses (flatten (map get-conjuncts-recursive goals)))))
     (match (Prolog.solve-aux program-clauses goal-clauses)
       ([true (some-sub sub)] (let ((variables (vars* goals))
                                     (bindings (list-zip variables (map sub variables))))
                                 [true (Prolog.transform-sub (make-sub bindings))]))

       (res res))))

(define (solve-all goals)
  (let ((program-clauses (get-all-horn-clauses goals))
        (goals (match goals ((some-list _) goals) (_ [goals])))
        (goal-clauses (get-all-clauses (flatten (map get-conjuncts-recursive goals))))
        (L (Prolog.solve-all-aux program-clauses goal-clauses)))
    (map (lambda (sub)
           (let ((variables (vars* goals))
                 (bindings (list-zip variables (map sub variables))))
             (Prolog.transform-sub (make-sub bindings))))
         L)))

(define (solve-N' goals N)
  (let ((program-clauses (get-all-horn-clauses goals))
        (goals (match goals ((some-list _) goals) (_ [goals])))
        (goal-clauses (get-all-clauses (flatten (map get-conjuncts-recursive goals))))
        (L (Prolog.solve-N-aux goal-clauses program-clauses N (- 1))))
    (map (lambda (sub)
           (let ((variables (vars* goals))
                 (bindings (list-zip variables (map sub variables))))
             (Prolog.transform-sub (make-sub bindings))))
         L)))

(Prolog.dont-print-info)

} # module Horn

extend-module Prolog {

(Prolog.dont-print-info)

(define (solve-goal g)
 (let ((_ (Prolog.dont-print-info))
        (_ ()))
    (Horn.solve g)))

(define auto-solve solve-goal)

(define (solve-goal-all g)
 (let ((_ (Prolog.dont-print-info)))
    (Horn.solve-all g)))

(define (defining-clauses g)
  (Horn.get-all-horn-clauses g))

(define (query-clauses g)
 (let ((g (match g
            ((some-list _) g)
            (_ [g]))))
    (Horn.get-all-clauses (flatten (map get-conjuncts-recursive g)))))
```

```

(define auto-solve-all solve-goal-all)

(define (solve-goal-N g N)
 (let ((_ (Prolog.dont-print-info)))
    (Horn.solve-N' g N)))

(define auto-solve-N solve-goal-N)

}

set-precedence Prolog.solve-goal 50
set-precedence Prolog.auto-solve 50

EOF
(load "lib/basic/prolog.ath")
```