```
(* upr.sml 470d *)
(* environments 186 *)
type name = string
type 'a env = (name * 'a) list
val emptyEnv = []

(* lookup and assignment of existing bindings *)
exception NotFound of name
fun find (name, []) = raise NotFound name
  | find (name, (n, v)::tail) = if name = n then v else find(name, tail)

(* adding new bindings *)
exception BindListLength
fun bind(name, v, rho) = (name, v) :: rho
fun bindList(n::vars, v::vals, rho) = bindList(vars, vals, bind(n, v, rho))
  | bindList([], [], rho) = rho
  | bindList _ = raise BindListLength
(* type declararations for consistency checking *)
val _ = op emptyEnv : 'a env
val _ = op find     : name * 'a env -> 'a
val _ = op bind     : name      * 'a      * 'a env -> 'a env
val _ = op bindList : name list * 'a list * 'a env -> 'a env
(* environments 604e *)
val tracer = ref (app print)
val _ = tracer := (fn _ => ())
fun trace l = !tracer l
(* abstract syntax and values 441a *)
datatype term = VAR     of string
              | LITERAL of int
              | APPLY   of string * term list
(* abstract syntax and values 441b *)
type goal = string * term list
(* abstract syntax and values 441c *)
datatype clause = :- of goal * goal list
infix 3 :-
(* abstract syntax and values 442 *)
datatype toplevel
  = ADD_CLAUSE of clause
  | QUERY      of goal list
  | USE        of string
(* abstract syntax and values 463a *)
(* string conversions 601a *)
fun termString (APPLY ("cons", [car, cdr])) =
      let fun tail (APPLY("cons", [car, cdr])) = ", " ^ termString car ^ tail
                                                                        cdr
            | tail (APPLY("nil",  [] ))        = "]"
            | tail x                           = "|" ^ termString x ^ "]"
      in  "[" ^ termString car ^ tail cdr
      end
  | termString (APPLY ("nil", [])) = "[]"
  | termString (APPLY (f, [])) = f
  | termString (APPLY (f, [x, y])) =
      if Char.isAlpha (hd (explode f)) then appString f x [y]
      else String.concat ["(", termString x, " ", f, " ", termString y, ")"]
  | termString (APPLY (f, h::t)) = appString f h t
  | termString (VAR v) = v
  | termString (LITERAL n) = if n < 0 then "-" ^ Int.toString (~n) else
                                                              Int.toString n
and appString f h t =
      String.concat (f :: "(" :: termString h ::
```

```
                  foldr (fn (t, tail) => ", " :: termString t :: tail) [")"]
                                                                       t)

(* string conversions 601b *)
fun goalString g = termString (APPLY g)
fun clauseString (g :- []) = goalString g
  | clauseString (g :- (h :: t)) =
      String.concat (goalString g :: " :- " :: goalString h ::
                     (foldr (fn (g, tail) => ", " :: goalString g :: tail)) [] t
                                                                             )
(* type declararations for consistency checking *)
val _ = op termString   : term   -> string
val _ = op goalString   : goal   -> string
val _ = op clauseString : clause -> string
(* clause database 463b *)
type database = clause list
val emptyDatabase = []
fun addClause (r, rs) = rs @ [r] (* must maintain order *)
fun potentialMatches (_, rs) = rs
(* substitution and unification 464b *)
(* substitution on terms 464a *)
exception LeftAsExercise
type subst = term -> term
infix 7 |-->
fun name |--> term = raise LeftAsExercise
fun idsubst term = term
(* type declararations for consistency checking *)
type database = database
val _ = op emptyDatabase    : database
val _ = op addClause        : clause * database -> database
val _ = op potentialMatches : goal * database -> clause list
(* type declararations for consistency checking *)
type subst = subst
val _ = op idsubst : subst
val _ = op |-->    : name * term -> subst
(* substitution and unification 464c *)
fun lift       theta (f, l)    = (f, map theta l)
fun liftClause theta (c :- ps) = (lift theta c :- map (lift theta) ps)
(* type declararations for consistency checking *)
val _ = op lift       : subst -> (goal   -> goal)
val _ = op liftClause : subst -> (clause -> clause)
(* substitution and unification 464d *)
type 'a set = 'a list
val emptyset = []
fun member x =
  List.exists (fn y => y = x)
fun insert (x, l) =
  if member x l then l else x::l
fun diff  (s1, s2) =
  List.filter (fn x => not (member x s2)) s1
fun union (s1, s2) = s1 @ diff (s2, s1)   (* preserves order *)
(* type declararations for consistency checking *)
type 'a set = 'a set
val _ = op emptyset : 'a set
val _ = op member   : ''a -> ''a set -> bool
val _ = op insert   : ''a     * ''a set  -> ''a set
val _ = op union    : ''a set * ''a set  -> ''a set
val _ = op diff     : ''a set * ''a set  -> ''a set
(* substitution and unification 465a *)
fun freevars t =
```

```
  let fun f(VAR v,     l) = insert (v, l)
        | f(LITERAL _, l) = l
        | f(APPLY(_, args), l) = foldl f l args
  in  rev (f(t, []))
  end
fun freevarsGoal goal = freevars (APPLY goal)
fun freevarsClause (c :- ps) =
  foldl (fn (p, f) => union (freevarsGoal p, f)) (freevarsGoal c) ps
(* type declararations for consistency checking *)
val _ = op freevars       : term   -> name set
val _ = op freevarsGoal   : goal   -> name set
val _ = op freevarsClause : clause -> name set
(* substitution and unification 465b *)
local
  val n = ref 1
in
  fun freshVar s = VAR ("_" ^ s ^ Int.toString (!n) before n := !n + 1)
(* type declararations for consistency checking *)
val _ = op freshVar : string -> term
end
(* substitution and unification 465c *)
fun freshen c =
  let val free     = freevarsClause c
      val fresh    = map freshVar free
      val renaming = ListPair.foldl (fn (free, fresh, theta) => theta o (free
                                                                         |--> fresh))
                       idsubst (free, fresh)
  in  liftClause renaming c
  end
(* substitution and unification 466 *)
(* unification 465d *)
exception Unify
fun unifyTerm (t1, t2) =
  raise LeftAsExercise
and unifyList (l1, l2) =
  raise LeftAsExercise
fun unify ((f, args), (f', args')) =
  if f = f' then unifyList (args, args') else raise Unify
(* type declararations for consistency checking *)
val _ = op freshen : clause -> clause
(* type declararations for consistency checking *)
val _ = op unify     : goal      * goal      -> subst
val _ = op unifyTerm : term      * term      -> subst
val _ = op unifyList : term list * term list -> subst
(* lexical analysis 605a *)
datatype token
  = UPPER     of string
  | LOWER     of string
  | SYMBOLIC  of string
  | INT_TOKEN of int
  | RESERVED  of string
(* lexical analysis 605b *)
fun tokenString (UPPER s) = s
  | tokenString (LOWER s) = s
  | tokenString (INT_TOKEN n) = Int.toString n
  | tokenString (SYMBOLIC  s) = s
  | tokenString (RESERVED  s) = s
(* lexical analysis 605c *)
val symbols = explode "!%^&*-+:=|~<>/?`$\\"   (* ask Russ Cox why there is a $
                                                                        here *)
```

```
fun isSymbol c = List.exists (fn c' => c' = c) symbols
fun isIdent  c = Char.isAlphaNum c orelse c = #"_"
fun isDelim  c = not (isIdent c orelse isSymbol c)
(* lexical analysis 605d *)
fun symbolic ":-" = RESERVED ":-"
  | symbolic "."  = RESERVED "."
  | symbolic "|"  = RESERVED "|"
  | symbolic "!"  = LOWER "!"
  | symbolic s    = SYMBOLIC s
fun lower "is" = RESERVED "is"
  | lower s    = LOWER s
(* lexical analysis 605e *)
fun anonymousVar () =
  case freshVar ""
    of VAR v => UPPER v
     | _ => let exception ThisCan'tHappen in raise ThisCan'tHappen end
(* lexical analysis 606a *)
exception Paren of char list  (* raise if found ) when reading input *)
exception PrematureEOF of string (* shows what we were reading ... *)
exception EOF                 (* raised by reader *)
exception SyntaxError of string
fun read nextline : string -> token list =
  let fun readmap f (result, unconsumed) = (f result, unconsumed)
      (* get is called only when s is guaranteed non-blank, non-comment, non-
                                                                        empty *)
      (* character-fiddling functions 570b *)
      fun isDelim c = List.exists (fn c' => c' = c) [#"(", #")", #" ", #"\t",
                                                     #"\n", #";"]
      (* character-fiddling functions 571a *)
      fun stringReader []     = NONE
        | stringReader (h::t) = SOME (h, t)
      val strton = Int.scan StringCvt.DEC stringReader
      (* type declararations for consistency checking *)
      val _ = op strton : char list -> (int * char list) option
      (* character-fiddling functions 571b *)
      fun skip s =
        case StringCvt.skipWS stringReader s
          of #";" :: s => skip (StringCvt.dropl (fn c => c <> #"\n")
                                                stringReader s)
           | s => s
      (* character-fiddling functions 571c *)
      fun nonwhite s =
        case skip s
          of [] => nonwhite (explode (nextline()))
           | s  => s
      fun isDelim  c = not (Char.isAlphaNum c orelse isSymbol c)
      exception Empty
      (*arg to get may be empty, but never begins with white space*)
      fun get [] = raise Empty
        | get (#"_"::nil)  = (anonymousVar (), nil)
        | get (chars as #"_"::c::s) =
            if isIdent c then
              raise SyntaxError ("names may not begin with underscores at " ^
                                                          implode chars)
            else
              (anonymousVar (), c::s)
        | get (#"/" :: #"*" :: s) = skipComment s
        | get (#"-" :: c :: s) =
            if Char.isDigit c then
               atom(c::s, [#"-"], implodeInt, Char.isDigit)
```

```
            else
                atom(#"-" :: c :: s, [], symbolic o implode, isSymbol)
        | get (c::s) =
            if      isDelim       c then (RESERVED (str c), s)
            else if Char.isLower  c then atom(c::s, [], lower    o implode,
                                                                   isIdent)
            else if Char.isUpper  c then atom(c::s, [], UPPER    o implode,
                                                                   isIdent)
            else if isSymbol      c then atom(c::s, [], symbolic o implode,
                                                                 isSymbol)
            else if Char.isDigit  c then atom(c::s, [], implodeInt,
                                                               Char.isDigit)
            else raise SyntaxError ("invalid initial character in `" ^ implode (
                                                                c::s) ^ "'")
      and atom(c::s, r, mkAtom, continue) =
            if continue c then atom(s, c::r, mkAtom, continue)
            else (mkAtom(rev r), c::s)
        | atom([],   r, mkAtom, continue) = (mkAtom(rev r), [])
      and implodeInt s =
        case strton s of SOME (n, []) => INT_TOKEN n
                       | _ => raise SyntaxError "this can't happen"
      and skipComment (#"*" :: #"/" :: s) = get (skip s)
        | skipComment (c :: s) = skipComment s
        | skipComment [] = skipComment (explode (nextline()))
      fun readline [] = []
        | readline s  =
             let val(p, u) = get s
             in  p :: readline(skip u)
             end handle Empty => []
  in  fn s => readline(skip (explode s))
  end
(* type declararations for consistency checking *)
val _ = op read : (unit -> string) -> string -> token list
(* parsing 602c *)
datatype concrete
  = BRACKET of string
  | CLAUSE  of goal * goal list option
  | GOALS  of goal list
(* parsing *)
fun toplevel _ = let exception Unimp in raise Unimp end
(* parsing 607c *)
fun synerror (t, nt, s) =
  let val msg = foldr (fn (t, l) => " " :: tokenString t :: l) ("; " :: s) t
      val msg =  "while parsing " :: nt :: " found" :: msg
  in  raise SyntaxError (concat msg)
  end
(* type declararations for consistency checking *)
val _ = op synerror : token list * string * string list -> 'a
(* parsing 608a *)
local
  (* lazy lists 608b *)
  structure Susp = struct
    datatype 'a contents = THUNK of unit -> 'a | FORCED of 'a
    type 'a susp = 'a contents ref
    fun delay f = ref (THUNK f)
    fun force cell =
      case !cell
        of THUNK f  => let val v = f() in v before cell := FORCED v end
         | FORCED v => v
    fun valueIn cell =
```

```
    case !cell
      of THUNK f  => NONE
       | FORCED v => SOME v
end
(* lazy lists 609a *)
datatype 'a llist
  = ::: of 'a * 'a llist
  | NEED_MORE of 'a llist Susp.susp
infixr 3 :::
fun fromLazy (h ::: t) = h :: fromLazy t
  | fromLazy (NEED_MORE l) = fromLazy(valOf(Susp.valueIn l)) handle Option =>
                                                                        []
fun toLazy more items =
  foldr op ::: (NEED_MORE (Susp.delay (fn () => toLazy more (more())))) items
(* lazy lists 610b *)
fun lforce f (NEED_MORE t) = lforce f (Susp.force t)
  | lforce f tokens        = f tokens
(* combinators for parsing \uprolog 609b *)
type 'a parser = token llist -> 'a * token llist
(* type declararations for consistency checking *)
val _ = op fromLazy : 'a llist -> 'a list
val _ = op toLazy   : (unit -> 'a list) -> 'a list -> 'a llist
(* type declararations for consistency checking *)
type 'a parser = 'a parser
(* combinators for parsing \uprolog 609c *)
exception E of string * string * token llist
fun fail nt why toks = raise E(nt, why, toks)

infix 0 ||
fun (p1 || p2) toks = p1 toks handle E _ => p2 toks
(* type declararations for consistency checking *)
val _ = op ||  : 'a parser * 'a parser -> 'a parser
(* combinators for parsing \uprolog 609d *)
fun !! p tokens =
  p tokens handle E (nt, why, tokens) =>
             synerror(fromLazy tokens, nt, [why])
(* type declararations for consistency checking *)
val _ = op !!  : 'a parser -> 'a parser
(* combinators for parsing \uprolog 609e *)
infix 5 --
fun (p1 -- p2) tokens =
  let val (x, tokens) = p1 tokens
      val (y, tokens) = p2 tokens
  in  ((x, y), tokens)
  end
(* type declararations for consistency checking *)
val _ = op -- : 'a parser * 'b parser -> ('a * 'b) parser
(* combinators for parsing \uprolog 610a *)
fun symbol (SYMBOLIC s:::tokens) = (s, tokens)
  | symbol tokens = fail "symbol" "symbol" tokens
fun upper (UPPER a ::: tokens) = (a, tokens)
  | upper tokens =
      fail "ident" "Identifier expected" tokens
fun lower (LOWER a ::: tokens) = (a, tokens)
  | lower tokens = fail "atom" "functor or predicate expected" tokens
fun int (INT_TOKEN a ::: tokens) = (a, tokens)
  | int tokens = fail "atom" "functor or predicate expected" tokens
fun $a (ts as RESERVED b ::: tokens) =
       if a = b then (a, tokens) else fail "unknown" ("expected " ^ a) ts
  | $a ts = fail "unknown" ("expected " ^ a) ts
```

```
(* combinators for parsing \uprolog 610c *)
val (lower, upper, int, symbol, $) =
  (lforce lower, lforce upper, lforce int, lforce symbol, lforce o $)
(* type declararations for consistency checking *)
val _ = op symbol : string parser
val _ = op upper  : string parser
val _ = op lower  : string parser
val _ = op int    : int    parser
val _ = op $ : string -> string parser
(* type declararations for consistency checking *)
val _ = op lforce   : ('a llist -> 'b) -> ('a llist -> 'b)
(* combinators for parsing \uprolog 610d *)
infix 3 >>
fun (p >> f) tokens =
  let val (x, tokens) = p tokens
  in  (f x, tokens)
  end
(* type declararations for consistency checking *)
val _ = op >> : 'a parser * ('a -> 'b) -> 'b parser
(* combinators for parsing \uprolog 610e *)
infix 6 $-- --$
fun (a $-- p) = ($a -- !!p >> #2)
fun (p --$ a) = (p -- !!($a) >> #1)
(* type declararations for consistency checking *)
val _ = op $-- : string    * 'a parser -> 'a parser
val _ = op --$ : 'a parser * string    -> 'a parser
(* combinators for parsing \uprolog 611a *)
fun empty tokens = ([], tokens)
fun repeat p tokens =
  (p -- lforce (repeat p) >> op :: || empty) tokens
fun commas p tokens =
  (p -- ("," $-- lforce (commas p) || empty) >> op ::) tokens
(* type declararations for consistency checking *)
val _ = op repeat   : 'a parser -> 'a list parser
val _ = op commas   : 'a parser -> 'a list parser
val _ = op empty    : 'a list parser
(* combinators for parsing \uprolog 611b *)
fun optional p =
  p >> SOME || empty >> (fn _ => NONE)
(* type declararations for consistency checking *)
val _ = op optional : 'a parser -> 'a option parser
(* combinators for parsing \uprolog 611c *)
fun notSymbol (tokens as SYMBOLIC s ::: _) =
      synerror(fromLazy tokens, "expression",
               ["arithmetic expressions must be parenthesized"])
  | notSymbol tokens = ((), tokens)
val notSymbol = lforce notSymbol : unit parser
(* combinators for parsing \uprolog 611d *)
val nilt = empty >> (fn _ => APPLY("nil", [])) : term parser
fun cons (x, y) = APPLY("cons", [x, y])
(* type declararations for consistency checking *)
val _ = op nilt : term parser
val _ = op cons : term * term -> term
(* combinators for parsing \uprolog 612a *)
fun term tokens = lforce
  (   "[" $-- (commas term -- ("|" $-- !!term || nilt)
              || empty -- nilt
              ) --$ "]"           >> (fn (elems, tail) => foldr cons tail
                                                                  elems)
  ||  atom -- "is" $-- !!term     >> (fn (a, t) => APPLY("is", [a, t]))
```

```
    ||  atom -- symbol -- !!atom -- !!notSymbol
                                    >> (fn (((l, f), r), _) => APPLY(f, [l, r]))
    ||  atom
    )
    tokens
  and atom tokens = lforce
    (   lower -- ("(" $-- commas (!!term) --$ ")" || empty)  >> APPLY
    ||  upper    >> VAR
    ||  int      >> LITERAL
    ||  "(" $-- term --$ ")"
    ||  fail "" "expected term"
    )
    tokens
  (* type declararations for consistency checking *)
  val _ = op term : term     parser
  val _ = op atom : term     parser
  (* combinators for parsing \uprolog 612b *)
  fun toGoal (APPLY g) = g
    | toGoal (VAR v)   =
        raise SyntaxError ("Variable " ^ v ^ " cannot be a predicate")
    | toGoal (LITERAL n) =
        raise SyntaxError ("Integer " ^ Int.toString n ^
                                                  " cannot be a predicate")
  val goal = (term || fail "" "expected goal") >> toGoal : goal parser
  (* type declararations for consistency checking *)
  val _ = op toGoal : term -> goal
  val _ = op goal   : goal parser
  (* combinators for parsing \uprolog 612c *)
  fun file (tokens as RESERVED "]" ::: _) =
        fail "filename" "found closing ]" tokens
    | file (t ::: tokens) = (tokenString t, tokens)
    | file (NEED_MORE _) = raise SyntaxError "expected `]' before end of line"
  val file = repeat (lforce file) >> String.concat : string parser

  val concrete
    =  "[" $-- file --$ "]" --$ "."             >> BRACKET
    ||  goal -- ":-" $-- commas goal --$ "."    >> (fn (g, gs) => CLAUSE (g,
                                                                  SOME gs))
    ||  commas goal --$ "."                     >> GOALS
  val concrete =
    fn tokens => !! (lforce concrete) tokens
                 handle SyntaxError s => synerror(fromLazy tokens, "concrete",
                                          ["\n ... preceding message was ", s]
                                                                        )
  (* type declararations for consistency checking *)
  val _ = op concrete : concrete parser
in
  fun getconcrete nextline tokens =
      let fun parse [] = []
            | parse tokens =
                let val (c, tokens) = concrete (toLazy nextline tokens)
                in  c :: parse (fromLazy tokens)
                end
      in  parse tokens
      end
(* type declararations for consistency checking *)
val _ = op getconcrete : (unit -> token list) -> token list -> concrete list
val _ = op concrete    : token llist -> concrete * token llist
val _ = op fromLazy    : 'a llist -> 'a list
val _ = op toLazy      : (unit -> 'a list) -> 'a list -> 'a llist
```

```
end
(* parsing *)
fun newparse reader = (* a bit bogus on prompting *)
  let fun tokens () = read reader (reader ())
  in  fn () => getconcrete tokens (tokens())
  end

fun cprint f [] = print "<??empty??>"
  | cprint f (h::t) = (print (f h); app (fn x => app print [", ", f x]) t)

fun printTop (BRACKET s) = app print ["[", s, "].\n"]
  | printTop (CLAUSE (g, rhs)) =
       ( print (goalString g)
       ; case rhs
           of NONE => ()
            | SOME gs => (print " :- "; cprint goalString gs)
       ; print ".\n"
       )
  | printTop (GOALS gs) = (cprint goalString gs; print ".\n")
(* readers 573b *)
type reader = unit -> string (* raises EOF *)
(* readers 573c *)
fun filereader fd () =
  let val line = TextIO.inputLine fd
  in  if size line = 0 then raise EOF else line
  end

fun stringsreader l =
  let val buffer = ref l
  in  fn () => case !buffer
                 of [] => raise EOF
                  | h :: t => h before buffer := t
  end
(* type declararations for consistency checking *)
val _ = op filereader    : TextIO.instream -> reader
val _ = op stringsreader : string list     -> reader
(* readers 574a *)
type topreader = { buffer    : toplevel list ref
                 , nextline  : unit -> string
                 , firstline : unit -> string
                 }
fun readtop (r as { buffer, nextline, firstline }) =
  case !buffer
    of h::t => h before buffer := t
     | []   => ( buffer := map toplevel (read nextline (firstline()))
               ; readtop r
               )
(* readers 574b *)
fun topreader (getline, prompt) =
  let fun promptIn prompt () = ( TextIO.output(TextIO.stdOut, prompt)
                               ; TextIO.flushOut(TextIO.stdOut)
                               ; getline ()
                               )
  in  if prompt then
        { buffer = ref [], nextline = promptIn "   ", firstline = promptIn "-> " }
      else
        { buffer = ref [], nextline = getline, firstline = getline }
  end
(* readers 574c *)
```

```
fun echoTag f x =
  let val line = f x
      val _ = if (String.substring (line, 0, 2) = ";#" handle _ => false) then
                print line
              else
                ()
  in  line
  end

fun echoBuf { buffer=b, nextline=n, firstline=f } =
 { buffer=b, nextline=echoTag n, firstline=echoTag f }

val topreader = fn args => echoBuf (topreader args)
(* readers 602b *)
datatype mode = QMODE | RMODE
(* readers 602d *)
type topreader = { buffer    : concrete list ref
                 , mode      : mode ref
                 , nextline  : unit -> string
                 , firstline : mode -> string
                 }
(* readers 603a *)
fun readtop (r as { buffer, nextline, firstline, mode }) : toplevel =
  let fun tokens () = read nextline (firstline (!mode))
      fun item () =
        case !buffer
          of h::t => h before buffer := t
           | []   => ( buffer := getconcrete tokens (tokens())
                     ; item ()
                     )
  in  case (item(), !mode)
        of (BRACKET "rule",   _) => (mode := RMODE; readtop r)
         | (BRACKET "fact",   _) => (mode := RMODE; readtop r)
         | (BRACKET "user",   _) => (mode := RMODE; readtop r)
         | (BRACKET "clause", _) => (mode := RMODE; readtop r)
         | (BRACKET "query",  _) => (mode := QMODE; readtop r)
         | (BRACKET s, _)        => USE s
         | (CLAUSE (g, ps),     RMODE) => ADD_CLAUSE (g :- getOpt(ps, []))
         | (CLAUSE (g, NONE),   QMODE) => QUERY [g]
         | (CLAUSE (_, SOME _), QMODE) =>
             raise SyntaxError ("You cannot enter a new clause in query mode; "
                                                                              ^
                                "type `[rule].' to change modes")
         | (GOALS l,   QMODE) => QUERY l
         | (GOALS [g], RMODE) => ADD_CLAUSE (g :- [])
         | (GOALS l,   RMODE) =>
             raise SyntaxError ("You cannot enter a query in clause mode; " ^
                                "type `[query].' to change modes")
  end
(* type declararations for consistency checking *)
val _ = op readtop : topreader -> toplevel
(* readers 603b *)
fun topreader' mode (getline, prompt) =
  let fun promptIn prompt () = ( TextIO.output(TextIO.stdOut, prompt)
                               ; TextIO.flushOut(TextIO.stdOut)
                               ; getline ()
                               )
      fun mprompt RMODE = promptIn "-> " ()
        | mprompt QMODE = promptIn "?- " ()
  in  if prompt then
```

```
        { buffer = ref [], mode = ref mode, nextline = echoTag (promptIn "   "),
          firstline = echoTag mprompt }
      else
        { buffer = ref [], mode = ref mode, nextline = echoTag getline,
          firstline = echoTag (fn _ => getline ()) }
  end
(* readers 603c *)
fun topreader x = topreader' RMODE x
(* implementation of [[use]] 193c *)
fun use readEvalPrint filename rho =
      let val fd = TextIO.openIn filename
          fun writeln s = app print [s, "\n"]
      in  readEvalPrint (topreader (filereader fd, false), writeln, writeln) rho
          before TextIO.closeIn fd
      end
(* primitives 469c *)
exception RuntimeError of string
fun eval (LITERAL n) = n
  | eval (APPLY ("+", [x, y])) = eval x  +  eval y
  | eval (APPLY ("*", [x, y])) = eval x  *  eval y
  | eval (APPLY ("-", [x, y])) = eval x  -  eval y
  | eval (APPLY ("/", [x, y])) = eval x div eval y
  | eval (APPLY ("-", [x]))    = ~ (eval x)
  | eval (APPLY (f, _))        =
      raise RuntimeError (f ^ " is not an arithmetic predicate " ^
                          "or is used with wrong arity")
  | eval (VAR v) = raise RuntimeError ("Used uninstantiated variable " ^ v ^
                                       " in arithmetic expression")
(* type declararations for consistency checking *)
val _ = op eval : term -> int
(* primitives 469d *)
fun is [x, e] succ fail = (succ (unifyTerm (x, LITERAL (eval e))) fail
                           handle Unify => fail())
  | is _      _    fail = fail ()
(* primitives 470a *)
fun compare name cmp [LITERAL n, LITERAL m] succ fail =
      if cmp (n, m) then succ idsubst fail else fail ()
  | compare name _ [_, _] _ _ =
      raise RuntimeError ("Used comparison " ^ name ^ " on non-integer term")
  | compare name _ _ _ _ =
      raise RuntimeError ("this can't happen---non-binary comparison?!")
(* tracing functions 494 *)
fun logSucc goal succ rho resume =
  ( app print ["SUCC: ", goalString goal, " becomes ", goalString (lift rho goal
                                                                   ), "\n"]
  ; succ rho resume
  )
fun logFail goal fail () =
  ( app print ["FAIL: ", goalString goal, "\n"]
  ; fail ()
  )
fun logResume goal resume () =
  ( app print ["REDO: ", goalString goal, "\n"]
  ; resume ()
  )
fun logSolve solve goal succ fail =
  ( app print ["START: ", goalString goal, "\n"]
  ; solve goal succ fail
  )
(* search 467a *)
```

```
fun 'a query database =
  let val builtins = foldl (fn ((n, p), rho) => bind (n, p, rho))
                     emptyEnv ((* primops :: 469b *)
                               ("print", fn args => fn succ => fn fail =>
                                          ( app (fn x => (print (termString x);
                                                          print " ")) args
                                          ; print "\n"
                                          ; succ (fn x => x) fail
                                          )) ::
                               (* primops :: 469e *)
                               ("is", is) ::
                               (* primops :: 470b *)
                               ("<",  compare "<"  op < ) ::
                               (">",  compare ">"  op > ) ::
                               ("=<", compare "=<" op <= ) ::
                               (">=", compare ">=" op >= ) ::
                               (* primops :: 470c *)
                               ("!",   fn _ => raise RuntimeError
                              "The cut (!) must be added to the interpreter") ::
                               ("not", fn _ => raise RuntimeError
                  "The predicate `not' must be added to the interpreter") :: [])
      fun solveOne (goal as (func, args)) succ fail =
            find(func, builtins) args succ fail
            handle NotFound _ =>
              let fun search [] = fail ()
                    | search (clause :: clauses) =
                        let fun resume() = search clauses
                            val conclusion :- premises = freshen clause
                            val theta = unify (goal, conclusion)
                        in  solveMany (map (lift theta) premises) theta succ
                                                                        resume
                        end
                        handle Unify => search clauses
(* type declararations for consistency checking *)
val _ = op query  : database -> goal list -> (subst -> (unit->'a) -> 'a) -> (
                                                          unit->'a) -> 'a
val _ = op solveOne  : goal               -> (subst -> (unit->'a) -> 'a) -> (
                                                          unit->'a) -> 'a
val _ = op solveMany : goal list -> subst -> (subst -> (unit->'a) -> 'a) -> (
                                                          unit->'a) -> 'a
val _ = op search    : clause list -> 'a
              in  search (potentialMatches (goal, database))
              end
      and solveMany []            theta succ fail = succ theta fail
        | solveMany (goal::goals) theta succ fail =
            solveOne goal
            (fn theta' => fn resume =>
               solveMany (map (lift theta') goals) (theta' o theta) succ resume)
            fail
  in  fn gs => solveMany gs (fn x => x)
  end
(* interaction 468c *)
fun showAndContinue prompt theta gs =
  let fun showVar v = app print [v, " = ", termString (theta (VAR v))]
      val vars = foldr union emptyset (map freevarsGoal gs)
  in  case vars
        of [] => false
         | h :: t => ( showVar h
                     ; app (fn v => (print "\n"; showVar v)) t
                     ; if prompt then
```

```
                        case explode (TextIO.inputLine TextIO.stdIn)
                          of #";" :: _ => (print "\n"; true)
                           | _ => false
                      else
                        (print "\n"; false)
                   )
  end
(* evaluation 467b *)
fun topeval prompt (t, database, echo) =
  case t
    of USE filename => (* read from file [[filename]] 468a *)
                       let fun try filename = use (readEvalPrint false) filename
                                                                        database
                       in  try filename handle IO.Io _ => try (filename ^ ".P")
                       end
     | ADD_CLAUSE c => addClause (c, database)
     | QUERY gs     => ((* query goals [[gs]] against [[database]] 468b *)
                        query database gs
                          (fn theta => fn resume =>
                             if showAndContinue prompt theta gs then resume()
                                                          else print "yes\n")
                          (fn () => print "no\n"); database)
(* evaluation 469a *)
and readEvalPrint prompt (reader, echo, errmsg) database =
  let fun loop database =
        let fun continue msg = (errmsg msg; loop database)
            fun finish() = database
        in  loop (topeval prompt (readtop reader, database, echo))
            handle EOF => finish()
                (* more read-eval-print handlers 194b *)
                | IO.Io {name, ...} => continue ("I/O error: " ^ name)
                | Paren _           => continue ("syntax error: extra )")
                | PrematureEOF s    => (errmsg ("Missing )? Got EOF reading (" ^
                                                                  s); finish())
                | SyntaxError msg   => continue ("syntax error: " ^ msg)
                (* more read-eval-print handlers 194c *)
                | Div               => continue "Division by zero"
                | Overflow          => continue "Arithmetic overflow"
                | RuntimeError msg  => continue ("run-time error: " ^ msg)
                | NotFound n        => continue ("variable " ^ n ^ " not found")
        end
  in  loop database
  end
(* type declararations for consistency checking *)
val _ = op topeval : bool -> toplevel * database * (string->unit) -> database
(* type declararations for consistency checking *)
val _ = op readEvalPrint : bool -> topreader * (string->unit) * (string->unit)
                                                    -> database -> database
(* Prolog command line 604b *)
fun runInterpreter noisy =
  let fun writeln s = app print [s, "\n"]
      val mode = if noisy then QMODE else RMODE
  in  ignore (readEvalPrint noisy (topreader' mode (filereader TextIO.stdIn,
                                                                    noisy),
                                   writeln, writeln) emptyDatabase)
  end
(* Prolog command line *)
fun copy file =
  let val reader = filereader file
      val parse = newparse reader
```

```
      fun go () = (app printTop (parse()); print "/**----**/\n"; go())
  in  go () handle EOF => ()
                 | SyntaxError s => app print ["Syntax error: ", s, "\n"]
  end
(* Prolog command line 604c *)
fun runmain ["-q"]          = runInterpreter false
  | runmain []              = runInterpreter true
  | runmain ["-copy"]       = copy TextIO.stdIn
  | runmain ("-trace" :: t) = (tracer := app print; runmain t)
  | runmain _  =
      TextIO.output(TextIO.stdErr, "Usage: " ^ CommandLine.name() ^ " [-q]\n")
(* Prolog command line *)
val _ = runmain (CommandLine.arguments())
```