Appendix J

Supporting code for μProlog

This Appendix is longer than many others:

- Even Prolog's simple syntax requires more code to parse than prefix-parenthesized syntax.
- In μProlog, as in C, a comment can span multiple lines, which means its lexical analyzer has to track source-code locations. This tracking needs extra code.
- A μProlog interpreter has two modes: rule mode and query mode. Tracking modes introduces additional complexity.

J.1 String conversions

This code converts terms, goals, and clauses to strings.

709 ⟨*string conversions* 709⟩≡ (551a) 710a▷

```
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) = String.map (fn #"~" => #"-" | c => c) (Int.toString n)
and appString f h t =
      String.concat (f :: "(" :: termString h ::
                     foldr (fn (t, tail) => ", " :: termString t :: tail) [")"] t)
```

710a ⟨*string conversions* 709⟩+≡ (551a) ◁709

```
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)
```

J.2 Lexical analysis

J.2.1 Tokens

μProlog has a more complex lexical structure than other languages. We have uppercase, lowercase, and symbolic tokens, as well as integers. It simplifies the parser if we distinguish reserved words and symbols using RESERVED. Finally, because a C-style μProlog comment can span multiple lines, we have to be prepared for the lexical analyzer to encounter end-of-file. Reading end of file needs to be distinguishable from failing to read a token, so I represent end of file by its own special token EOF.

710b ⟨*lexical analysis* 710b⟩≡ (559a) 710c▷

type token

```
datatype token
  = UPPER     of string
  | LOWER     of string
  | SYMBOLIC  of string
  | INT_TOKEN of int
  | RESERVED  of string
  | EOF
```

We need to print tokens in error messages.

710c ⟨*lexical analysis* 710b⟩+≡ (559a) ◁710b 710d▷

```
fun tokenString (UPPER s)     = s
  | tokenString (LOWER s)     = s
  | tokenString (INT_TOKEN n) = if n < 0 then "-" ^ Int.toString (~n)
                                else Int.toString n
  | tokenString (SYMBOLIC  s) = s
  | tokenString (RESERVED  s) = s
  | tokenString EOF           = "<end-of-file>"
```

We need to identify literals for the parser. The treatment of integer literals is a bit dodgy, but they shouldn't be used for parsing.

710d ⟨*lexical analysis* 710b⟩+≡ (559a) ◁710c 711b▷

```
fun isLiteral s t = (s = tokenString t)
⟨support for streams, lexical analysis, and parsing 644⟩
```

APPLY 529a
termString 709

J.2.2 Classification of characters

The other languages in this book treat only parentheses, digits, and semicolons specially. But in Prolog, we distinguish two kinds of names: symbolic and alphanumeric. A symbolic name like `+` is used differently from an alphanumeric name like `add1`. This difference is founded on a different classification of characters. In μProlog, every character is either a symbol, an alphanumeric, a space, or a delimiter.

711a ⟨*character-classification functions for μProlog* 711a⟩≡ (711d)

```
val symbols = explode "!%^&*-+:=|~<>/?‘$\\"
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)
```

J.2.3 Reserved words and anonymous variables

Tokens formed from symbols or from lower-case letters are usually symbolic, but sometimes they are reserved words. And because the cut is nullary, not binary, it is treated as an ordinary symbol, just like any other nullary predicate.

711b ⟨*lexical analysis* 710b⟩+≡ (559a) ◁710d 711c▷

```
fun symbolic ":-" = RESERVED ":-"
  | symbolic "."  = RESERVED "."
  | symbolic "|"  = RESERVED "|"
  | symbolic "!"  = LOWER "!"
  | symbolic s    = SYMBOLIC s
fun lower "is" = RESERVED "is"
  | lower s    = LOWER s
```

A variable consisting of a single underscore gets converted to a unique "anonymous" variable.

711c ⟨*lexical analysis* 710b⟩+≡ (559a) ◁711b 711d▷

```
fun anonymousVar () =
  case freshVar ""
    of VAR v => UPPER v
     | _ => let exception ThisCan’tHappen in raise ThisCan’tHappen end
```

J.2.4 Converting characters to tokens

We consume a stream of characters, intersperse with `EOL` (end-of-line) markers. We must product a stream of tokens. And unlike our other lexers, the μProlog lexer must produce *located* tokens, i.e., tokens that are tagged with source-code locations. The location corresponding to the start of the character stream is passed as a parameter to `tokenAt`.

freshVar	553b
LOWER	710b
RESERVED	710b
SYMBOLIC	710b
UPPER	710b
VAR	529a

711d ⟨*lexical analysis* 710b⟩+≡ (559a) ◁711c

```
local
  ⟨character-classification functions for μProlog 711a⟩
  ⟨lexical utility functions for μProlog 712a⟩
in
  ⟨lexical analyzers for for μProlog 712d⟩
end
```

Utility functions underscore and int make sure that an underscore or a sequence of digits, respectively, is never followed by any character that might be part of an alphanumeric identifier. When either of these functions succeeds, it returns an appropriate token.

712a ⟨*lexical utility functions for μProlog* 712a⟩≡ (711d) 712b ▷

```
underscore : char      -> char list -> token error
int        : char list -> char list -> token error
```

```
fun underscore _ [] = OK (anonymousVar ())
  | underscore c cs = ERROR ("name may not begin with underscore at " ^
                              implode (c::cs))

fun int cs [] = intFromChars cs >>=+ INT_TOKEN
  | int cs ids =
      ERROR ("integer literal " ^ implode cs ^
             " may not be followed by '" ^ implode ids ^ "'")
```

Utility function unrecognized is called when the lexical analyzer cannot recognize a sequence of characters. If the sequence is empty, it means there's no token. If anything else happens, an error has occurred.

712b ⟨*lexical utility functions for μProlog* 712a⟩+≡ (711d) ◁712a 712c ▷

```
unrecognized : char list error -> ('a error * 'a error stream) option
```

```
fun unrecognized (ERROR _) = let exception Can'tHappen in raise Can'tHappen end
  | unrecognized (OK cs) =
      case cs
        of []        => NONE
         | #";" :: _ => let exception Can'tHappen in raise Can'tHappen end
         | _ =>
             SOME (ERROR ("invalid initial character in `" ^ implode cs ^ "'"), EOS)
```

When a lexical analyzer runs out of characters on a line, it calls nextline to compute the location of the next line.

712c ⟨*lexical utility functions for μProlog* 712a⟩+≡ (711d) ◁712b

```
fun nextline (file, line) = (file, line+1)
```

```
nextline : srcloc -> srcloc
```

μProlog must be aware of the end of an input line. Lexical analyzers char and eol recognize a character and the end-of-line marker, respectively.

⟨*lexical analyzers for for μProlog* 712d⟩≡ (711d) 713a ▷

```
type 'a prolog_lexer
char : char prolog_lexer
eol  : unit prolog_lexer
```

```
type 'a prolog_lexer = (char inline, 'a) xformer
fun char chars =
  case streamGet chars
    of SOME (INLINE c, chars) => SOME (OK c, chars)
     | _ => NONE
fun eol chars =
  case streamGet chars
    of SOME (EOL _, chars) => SOME (OK (), chars)
     | _ => NONE
```

>>=+ 652a
anonymousVar 711c
EOL 662a
EOS 647a
ERROR 651a
INLINE 662a
INT_TOKEN 710b
intFromChars 660b
OK 651a
streamGet 647b

Functions charEq and manySat provide general tools for recognizing characters and sequences of characters. Lexers whitespace and intChars handle two common cases.

713a ⟨*lexical analyzers for for μProlog* 712d⟩+≡ (711d) ◁712d 713b▷

```
charEq     : char -> char prolog_lexer
manySat    : (char -> bool) -> char list prolog_lexer
whitespace : char list prolog_lexer
intChars   : char list prolog_lexer
```

```
fun charEq c =
  sat (fn c' => c = c') char
fun manySat p =
  many (sat p char)

val whitespace =
  manySat Char.isSpace
val intChars =
  (curry op :: <$> charEq #"-" <|> pure id) <*> many1 (sat Char.isDigit char)
```

An ordinary token is an underscore, delimiter, integer literal, symbolic name, or alphanumeric name. Uppercase and lowercase names produce different tokens.

713b ⟨*lexical analyzers for for μProlog* 712d⟩+≡ (711d) ◁713a 713c▷

```
ordinaryToken : token prolog_lexer
```

```
val ordinaryToken =
      underscore            <$> charEq #"_" <*>! manySat isIdent
  <|> (RESERVED o str)      <$> sat isDelim char
  <|> int                   <$> intChars    <*>! manySat isIdent
  <|> (symbolic o implode)  <$> many1 (sat isSymbol char)
  <|> curry (lower o implode o op ::) <$> sat Char.isLower char <*> manySat isIdent
  <|> curry (UPPER o implode o op ::) <$> sat Char.isUpper char <*> manySat isIdent
  <|> unrecognized o fst o valOf o many char
```

We need two main lexical analyzers that keep track of source locations: tokenAt produces tokens, and skipComment skips comments. They are mutually recursive, and in order to delay the recursive calls until a stream is supplied, each definition has an explicit cs argument, which contains a stream of inline characters.

713c ⟨*lexical analyzers for for μProlog* 712d⟩+≡ (711d) ◁713b

```
tokenAt     : srcloc -> token located prolog_lexer
skipComment : srcloc -> srcloc -> token located prolog_lexer
```

```
fun tokenAt loc cs =  (* eta-expanded to avoid infinite regress *)
  (whitespace *> (   charEq #"/" *> charEq #"*" *> skipComment loc loc
                 <|> charEq #";" *> many char *> eol *> tokenAt (nextline loc)
                 <|>                              eol *> tokenAt (nextline loc)
                 <|> (loc, EOF) <$ eos
                 <|> pair loc <$> ordinaryToken
                 )) cs
and skipComment start loc cs =
  (   charEq #"*" *> charEq #"/" *> tokenAt loc
  <|> char *> skipComment start loc
  <|> eol  *> skipComment start (nextline loc)
  <|> id <$>! pure (ERROR ("end of file looking for */ to close comment in " ^
                           srclocString start))
  ) cs
```

J.3 Parsing

J.3.1 Utilities for parsing μProlog

714a ⟨*parsing* 714a⟩≡ (559a) 714b▷

```
symbol : string parser
upper  : string parser
lower  : string parser
int    : int    parser
```

```
val symbol = (fn SYMBOLIC  s => SOME s | _ => NONE) <$>? token
val upper  = (fn UPPER     s => SOME s | _ => NONE) <$>? token
val lower  = (fn LOWER     s => SOME s | _ => NONE) <$>? token
val int    = (fn INT_TOKEN n => SOME n | _ => NONE) <$>? token
```

We use these combinators to define the grammar from Figure 11.2. We use `notSymbol` to ensure that a term like `3 + X` is not followed by another symbol. This means we don't parse such terms as `3 + X + Y`.

714b ⟨*parsing* 714a⟩+≡ (559a) ◁714a 714c▷

```
notSymbol : unit parser
```

```
val notSymbol =
  symbol <!> "arithmetic expressions must be parenthesized" <|>
  pure ()
```

Parser `nilt` uses the empty list of tokens to represent the empty list of terms. It needs an explicit type constraint to avoid falling afoul of the value restriction on polymorphism. Function `cons` combines two terms, which is useful for parsing lists.

714c ⟨*parsing* 714a⟩+≡ (559a) ◁714b 714d▷

```
nilt : term parser
cons : term * term -> term
```

```
fun nilt tokens = pure (APPLY ("nil", [])) tokens
fun cons (x, xs) = APPLY ("cons", [x, xs])
```

Here is one utility function `commas`, plus renamings of three other functions.

714d ⟨*parsing* 714a⟩+≡ (559a) ◁714c 715a▷

```
variable        : string parser
binaryPredicate : string parser
functr          : string parser
commas : 'a parser -> 'a list parser
```

```
val variable        = upper
val binaryPredicate = symbol
val functr          = lower
fun commas p =
  curry op :: <$> p <*> many ("," >-- p)
```

I have to spell "functor" without the "o" because in Standard ML, `functor` is a reserved word.

J.3.2 Parsing terms, atoms, and goals

We're now ready to parse μProlog. The grammar is based on the grammar from Figure 11.2 on page 528, except that I'm using named function to parse atoms, and I use some specialized tricks to organize the grammar. Concrete syntax is not for the faint of heart.

715a ⟨*parsing* 714a⟩+≡ (559a) ◁714d 715b▷

```
term   : term parser
atom   : term parser
commas : 'a parser -> 'a list parser
```

```
fun term tokens =
  (   "[" >-- ((fn elems => fn tail => foldr cons tail elems) <$>
                    commas term <*> ("|" >-- (term <?> "list element") <|> nilt)
              <|> nilt
              ) --< "]"
  <|> (fn a => fn t => APPLY ("is", [a, t])) <$> atom --< "is" <*> (term <?> "term")
  <|> (fn l => fn f => fn r => APPLY (f, [l, r])) <$>
      atom <*> binaryPredicate <*> (atom <?> "atom") <* notSymbol
  <|> atom
  )
  tokens
and atom tokens =
  (   curry APPLY <$> functr <*> (   "(" >-- commas (term <?> "term") --< ")"
                                <|> pure []
                                )
  <|> VAR     <$> variable
  <|> LITERAL <$> int
  <|> "(" >-- term --< ")"
  )
  tokens
```

Terms and goals shared the same concrete syntax but different abstract syntax. Every goal can be interpreted as a term, but not every term can be interpreted as a goal.

715b ⟨*parsing* 714a⟩+≡ (559a) ◁715a 716a▷

```
asGoal : srcloc -> term -> goal error
goal   : goal parser
```

```
fun asGoal _   (APPLY g) = OK g
  | asGoal loc (VAR v)   =
      errorAt ("Variable " ^ v ^ " cannot be a predicate") loc
  | asGoal loc (LITERAL n) =
      errorAt ("Integer " ^ Int.toString n ^ " cannot be a predicate") loc

val goal = asGoal <$> srcloc <*>! term
```

J.3.3 Recognizing concrete syntax using modes

I put together the μProlog parser in three layers. The bottom layer is the concrete syntax itself. For a moment let's ignore the *meaning* of μProlog's syntax and look only at what can appear. At top level, we might see

- A string in brackets
- A clause containing a `:-` symbol
- A list of one or more goals separated by commas

716a ⟨*parsing* 714a⟩+≡ (559a) ◁715b 716b▷

type concrete
concrete : concrete parser

```
datatype concrete
  = BRACKET of string
  | CLAUSE  of goal * goal list option
  | GOALS   of goal list

val notClosing = sat (fn RESERVED "]" => false | _ => true) token

val concrete =
     (BRACKET o concat o map tokenString) <$> "[" >-- many notClosing --< "]"
 <|> curry CLAUSE <$> goal <*> ":-" >-- (SOME <$> commas goal)
 <|> GOALS <$> commas goal
```

In most cases, we know what these things are supposed to mean, but there's one case in which we don't: a phrase like "`color(yellow).`" could be either a clause *or* a query. To know which is meant, we have to know the *mode*. In other words, the mode distinguishes `CLAUSE(g, NONE)` from `GOALS [g]`. A parser may be in either query mode or rule (clause) mode, and each mode has its own prompt.

716b ⟨*parsing* 714a⟩+≡ (559a) ◁716a 716c▷

type mode
mprompt : mode -> string

```
datatype mode = QMODE | RMODE

fun mprompt RMODE = "-> "
  | mprompt QMODE = "?- "
```

--<	664c
<$>	653c
<*>	653b
<|>	654b
>--	664c
commas	714d
type cq	530
curry	654a
type goal	529b
goal	715b
many	657d
RESERVED	710b
sat	656b
token	663a
tokenString	710c

The concrete syntax normally means a clause or query, which is denoted by the syntactic nonterminal symbol *clause-or-query* and represented by an ML value of type `cq` (see chunk 530 in Chapter 11). But particular concrete syntax, such as "`[rule].`" or "`[query].`," can be an instruction to change to a new mode. The middle layer of μProlog's parser produces a value of type `cq_or_mode`, which is defined as follows:

⟨*parsing* 714a⟩+≡ (559a) ◁716b 717a▷

type cq_or_mode

```
datatype cq_or_mode
  = CQ of cq
  | NEW_MODE of mode
```

The next level of μProlog's parser interpreters a concrete value according to the mode. All BRACKET values are interpreted in the same way regardless of mode, but clauses and especially GOALS are interpreted differently in rule mode and in query mode.

717a ⟨*parsing* 714a⟩+≡ (559a) ◁716c 717b▷

interpretConcrete : mode -> concrete -> cq_or_mode error

```
fun interpretConcrete mode =
  let val (newMode, cq, err) = (OK o NEW_MODE, OK o CQ, ERROR)
  in  fn c =>
        case (mode, c)
          of (_, BRACKET "rule")     => newMode RMODE
           | (_, BRACKET "fact")     => newMode RMODE
           | (_, BRACKET "user")     => newMode RMODE
           | (_, BRACKET "clause")   => newMode RMODE
           | (_, BRACKET "query")    => newMode QMODE
           | (_, BRACKET s)          => cq (USE s)
           | (RMODE, CLAUSE (g, ps)) => cq (ADD_CLAUSE (g :- getOpt (ps, [])))
           | (RMODE, GOALS [g])      => cq (ADD_CLAUSE (g :- []))
           | (RMODE, GOALS _ ) =>
                err ("You cannot enter a query in clause mode; " ^
                     "to change modes, type `[query].'")
           | (QMODE, GOALS gs)          => cq (QUERY gs)
           | (QMODE, CLAUSE (g, NONE))  => cq (QUERY [g])
           | (QMODE, CLAUSE (_, SOME _)) =>
                err ("You cannot enter a new clause in query mode; " ^
                     "to change modes, type `[rule].'")
  end
```

Parser cq_or_mode m parses a concrete according to mode m. If it sees something it doesn't recognize, it emits an error message and skips ahead until it sees a dot or the end of the input. Importantly, this parser never fails: it always returns either a cq_or_mode value or an error message.

717b ⟨*parsing* 714a⟩+≡ (559a) ◁717a 718a▷

cq_or_mode : mode -> cq_or_mode parser

```
val skippable =
  (fn SYMBOLIC "." => NONE | EOF => NONE | t => SOME t) <$>? token

fun badConcrete (loc, skipped) last =
  ERROR (srclocString loc ^ ": expected clause or query; skipping" ^
         concat (map (fn t => " " ^ tokenString t) (skipped @ last)))

fun cq_or_mode mode = interpretConcrete mode <$>!
  (   concrete --< "."
  <|> badConcrete <$> @@ (many  skippable) <*>! ([RESERVED "."] <$ literal ".")
  <|> badConcrete <$> @@ (many1 skippable) <*>! pure []  (* skip to EOF *)
  )
```

--<	664c
<$>	653c
<$>!	658c
<$>?	657a
<*>!	658c
<|>	654b
ADD_CLAUSE	530
BRACKET	716a
CLAUSE	716a
concrete	716a
CQ	716c
EOF	710b
ERROR	651a
GOALS	716a
literal	664b
many	657d
many1	658a
NEW_MODE	716c
OK	651a
pure	653a
QMODE	716b
QUERY	530
RESERVED	710b
RMODE	716b
srclocString	661a
SYMBOLIC	710b
token	663a
tokenString	710c
USE	530

J.3.4 Reading clauses and queries while tracking locations and modes

All the other languages in this book produce a stream of definitions using the `reader` function from page 669. We can't reuse that function because it doesn't tag tokens with locations and it doesn't keep track of modes. Instead, I define a somewhat more complex function, `prologReader`, below. At the core of `prologReader` is function `getCq`.

718a ⟨*parsing* 714a⟩+≡ (559a) ◁717b

```
prologReader : bool -> mode -> string * string stream -> cq stream
type read_state = string * mode * token located inline stream
getCq : read_state -> (cq * read_state) option
```

```
fun prologReader noisy initialMode (name, lines) =
  let val (ps1, ps2) = (mprompt initialMode, "   ")
      val thePrompt = ref ""  (* no prompt unless noisy *)
      val setPrompt = if noisy then (fn s => thePrompt := s) else (fn _ => ())

      type read_state = string * mode * token located inline stream
      ⟨utility functions for prologReader 718b⟩

      val lines = preStream (fn () => print (!thePrompt), echoTagStream lines)

      val chars =
        streamConcatMap
        (fn (loc, s) => streamOfList (map INLINE (explode s) @ [EOL (snd loc)]))
        (locatedStream (name, lines))

      fun getLocatedToken (loc, chars) =
        (case tokenAt loc chars
           of SOME (OK (loc, t), chars) => SOME (OK (loc, t), (loc, chars))
            | SOME (ERROR msg,   chars) => SOME (ERROR msg,   (loc, chars))
            | NONE => NONE
        ) before setPrompt ps2

      val tokens = stripErrors (streamOfUnfold getLocatedToken ((name, 1), chars))

  in  streamOfUnfold getCq (ps1, initialMode, streamMap INLINE tokens)
  end
```

echoTagStream 667b
EOF 710b
EOL 662a
ERROR 651a
getCq 719b
INLINE 662a
locatedStream 661d
mprompt 716b
OK 651a
preStream 649a
snd 654a
type stream 647a
streamConcatMap 650d
streamGet 647b
streamMap 649c
streamOfList 647c
streamOfUnfold 648d
stripErrors 668a
tokenAt 713c

The application of `streamMap INLINE` may look very strange, but many of the utility functions from Appendix D expect a stream of tokens tagged with `INLINE`. Even though we don't really need the `INLINE` for μProlog, it is easier to use a meaningless `INLINE` than it is to rewrite big chunks of Appendix D.

Function `getCq` uses `startsWithEOF` to check if the input stream has no more tokens.

718b ⟨*utility functions for* prologReader 718b⟩≡ (718a) 719a▷

```
fun startsWithEOF tokens =
  case streamGet tokens
    of SOME (INLINE (_, EOF), _) => true
     | _ => false
```

```
startsWithEOF : token located inline stream -> bool
```

If getCq detects an error, it skips tokens in the input up to and including the next dot.

719a ⟨*utility functions for* prologReader 718b⟩+≡ (718a) ◁718b 719b▷

```
                skipPastDot : token located inline stream -> token located inline stream
fun skipPastDot tokens =
  case streamGet tokens
    of SOME (INLINE (_, RESERVED "."), tokens) => tokens
     | SOME (INLINE (_, EOF), tokens) => tokens
     | SOME (_, tokens) => skipPastDot tokens
     | NONE => tokens
```

Function getCq tracks the prompt, the mode, and the remaining unread tokens, which together form the read_state. It also, when called, sets the prompt.

719b ⟨*utility functions for* prologReader 718b⟩+≡ (718a) ◁719a

```
fun getCq (ps1, mode, tokens) =     getCq : read_state -> (cq * read_state) option
  ( setPrompt ps1
  ; if startsWithEOF tokens then
      NONE
    else
      case cq_or_mode mode tokens
        of SOME (OK (CQ cq),         tokens) => SOME (cq, (ps1, mode, tokens))
         | SOME (OK (NEW_MODE mode), tokens) => getCq (mprompt mode, mode, tokens)
         | SOME (ERROR msg,          tokens) =>
                                         ( errorln ("error: " ^ msg)
                                         ; getCq (ps1, mode, skipPastDot tokens)
                                         )
         | NONE => ⟨fail epically with a diagnostic about tokens 719c⟩
  )
```

Parser cq_or_mode is always supposed to return something.

719c ⟨*fail epically with a diagnostic about* tokens 719c⟩≡ (719b)

```
let exception ThisCan'tHappenCqParserFailed
    val tokensStrings = map (fn t => " " ^ tokenString t) o valOf o peek (many token)
    val _ = app print (tokensStrings tokens)
in  raise ThisCan'tHappenCqParserFailed
end
```

J.4 Command line

μProlog's command-line processor differs from our other interpreters, because it has to deal with modes. In noisy state it starts waiting for a query, but in quiet state it waits for a rule.

719d ⟨*Prolog command line* 719d⟩≡ (559a) 720a▷

```
fun runInterpreter noisy =
  let fun writeln s = app print [s, "\n"]
      fun errorln s = TextIO.output (TextIO.stdErr, s ^ "\n")
      val mode = if noisy then QMODE else RMODE
      val cqs  =
        prologReader noisy mode ("standard input", streamOfLines TextIO.stdIn)
  in  ignore (streamFold (evalPrint noisy (writeln, errorln)) emptyDatabase cqs)
  end
```

The `-q` option is as in other interpreters, and the `-trace` option turns on tracing.

720a ⟨*Prolog command line* 719d⟩+≡ (559a) ◁719d 720b▷

```
fun runmain ["-q"]           = runInterpreter false
  | runmain []               = runInterpreter true
  | runmain ("-trace" :: t) = (tracer := app print; runmain t)
  | runmain _  =
      TextIO.output (TextIO.stdErr, "Usage: " ^ CommandLine.name() ^ " [-q]\n")
```

720b ⟨*Prolog command line* 719d⟩+≡ (559a) ◁720a

```
val _ = runmain (CommandLine.arguments())
```

Tracing code is helpful for debugging.

720c ⟨*environments* 720c⟩≡ (224 559a)

```
val tracer = ref (app print)
val _ = tracer := (fn _ => ())
fun trace l = !tracer l
```

runInterpreter 719d