MLとProlog

h_sakurai

MLとPrologの話をします

- Prolog入門
- PrologでML
- MLでProlog
- PrologでProlog

Shebang,演算子,lookup等

```
#!/usr/bin/env swipl --toplevel=halt --stand_alone=true -q
:- style_check(-singleton).
:- set_prolog_flag(double_quotes,codes).
:- initialization(main).
:- op(1200, xfx, [ -- ]).
:- op(910, xfx, [ ⊢ ]).
:- op(900, xfx, [ ⇓, : ]).
:- op(892, xfy, [ then, else ]).
:- op(891, xfy, [ in ]).
:- op(890, fx, [ letrec, let, if ]).
:- op(888, xfy, ::).
:- op(500, yfx, $).
:- set_prolog_flag(report_error,true).
:- set_prolog_flag(unknown,error).

bool(true). bool(false).

lookup((Γ, X :V ), X : V).
lookup((Γ, X1:V1), X : V) :- X1\==X, lookup(Γ, X : V).

term_expansion(A -- B, B :- A).
```

評価規則

```
% int, bool, if

integer(I),!
--%------------------------------------ (E-Int)
C ⊢ I ⇓ I.

bool(B),!
--%------------------------------------ (E-Bool)
C ⊢ B ⇓ B.

C ⊢ E1 ⇓ true, C ⊢ E2 ⇓ V
--%------------------------------------ (E-IfTrue)
C ⊢ if E1 then E2 else E3 ⇓ V.

C ⊢ E1 ⇓ false, C ⊢ E3 ⇓ V
--%------------------------------------ (E-IfFalse)
C ⊢ if E1 then E2 else E3 ⇓ V.

% 二項演算子

C ⊢ E1 ⇓ V1, C ⊢ E2 ⇓ V2, V is V1 + V2,!
--%------------------------------------ (E-Plus)
C ⊢ E1 + E2 ⇓ V.

C ⊢ E1 ⇓ V1, C ⊢ E2 ⇓ V2, V is V1 - V2,!
--%------------------------------------ (E-Minus)
C ⊢ E1 - E2 ⇓ V.

C ⊢ E1 ⇓ V1, C ⊢ E2 ⇓ V2,
(V1 < V2, V = true; V = false),!
--%------------------------------------ (E-Lt)
C ⊢ E1 < E2 ⇓ V.
```

```
% Let,変数,関数

C ⊢ E1 ⇓ V1, (C, X:V1) ⊢ E2 ⇓ V2
--%------------------------------------ (E-Let)
C ⊢ let X = E1 in E2 ⇓ V2.

atom(X),!, lookup(C, X : V),!
--%------------------------------------ (E-Var)
C ⊢ X ⇓ V.

!
--%------------------------------------ (E-Fun)
C ⊢ (X -> E) ⇓ (C ⊢ X -> E).
```

評価規則

```
% 組み込み関数

C ⊢ E ⇓ V,
string_codes(S, V),!,write(S),!
--%------------------------------------ (E-AppPrintString)
C ⊢ (print_string $ E) ⇓ V.

C ⊢ E ⇓ V, number_string(I, V),!
--%------------------------------------ (E-AppIntOfString)
C ⊢ (int_of_string $ E) ⇓ I.

C ⊢ E ⇓ V,
string_codes(S, V),!,write(S),nl,!
--%------------------------------------ (E-AppPrintlnString)
C ⊢ (println_string $ E) ⇓ V.

C ⊢ E ⇓ V,write(V),nl,!
--%------------------------------------ (E-AppPrintlnInt)
C ⊢ (println_int $ E) ⇓ V.

C ⊢ E ⇓ V,write(V),nl,!
--%------------------------------------ (E-AppPrintlnBool)
C ⊢ (println_bool $ E) ⇓ V.

% 関数とLet Rec

C ⊢ E1 ⇓ (C2 ⊢ X -> E0), C ⊢ E2 ⇓ V2,
(C2,X:V2) ⊢ E0 ⇓ V
--%------------------------------------ (E-App)
C ⊢ (E1 $ E2) ⇓ V.

(C,X:V1) ⊢ E1 ⇓ V1,(C,X:V1) ⊢ E2 ⇓ V2
--%------------------------------------ (E-LetRec)
C ⊢ (letrec X = E1 in E2) ⇓ V2.
```

```
% リストとパターンマッチ

C ⊢ E1 ⇓ V, C ⊢ E2 ⇓ V2
--%------------------------------------ (E-Cons)
C ⊢ (E1::E2) ⇓ [V|V2].

C ⊢ E1 ⇓ V, C ⊢ E2 ⇓ V2
--%------------------------------------ (E-Cons2)
C ⊢ [E1|E2] ⇓ [V|V2].

!
--%------------------------------------ (E-Nil)
C ⊢ [] ⇓ [].

C ⊢ E1 ⇓ [], C ⊢ E2 ⇓ V
--%------------------------------------ (E-MatchNil)
C ⊢ match(E1 | [] -> E2 | _) ⇓ V.

C ⊢ E1 ⇓ [V1|V2],
((C,X : V1),Y : V2) ⊢ E3 ⇓ V
--%------------------------------------ (E-MatchCons)
C ⊢ match(E1 | _ | X::Y->E3) ⇓ V.
●
```

型推論

```
% int,bool,if,二項演算子

integer(I)
--%------------------------------------ (T-Int)
Γ ⊢ I : int.

bool(B)
--%------------------------------------ (T-Bool)
Γ ⊢ B : bool.

Γ ⊢ E1 : bool, Γ ⊢ E2 : T, Γ ⊢ E3 : T
--%------------------------------------ (T-If)
Γ ⊢ (if E1 then E2 else E3) : T.

Γ ⊢ E1 : int, Γ ⊢ E2 : int
--%------------------------------------ (T-Plus)
Γ ⊢ E1 + E2 : int.

Γ ⊢ E1 : int, Γ ⊢ E2 : int
--%------------------------------------ (T-Minus)
Γ ⊢ E1 - E2 : int.

Γ ⊢ E1 : int, Γ ⊢ E2 : int
--%------------------------------------ (T-Lt)
Γ ⊢ (E1 < E2) : bool.

% 変数と関数とLet

atom(X), lookup(Γ, X : T)
--%------------------------------------ (T-Var)
Γ ⊢ X : T.

(Γ,X:T) ⊢ E : T2
--%------------------------------------ (T-Fun)
Γ ⊢ (X -> E) : (T -> T2).

Γ ⊢ E1 : (T2 -> T), Γ ⊢ E2 : T2
--%------------------------------------ (T-App)
Γ ⊢ E1 $ E2 : T.

Γ ⊢ E1 : T1, (Γ,X:T1) ⊢ E2 : T2
--%------------------------------------ (T-Let)
Γ ⊢ (let X = E1 in E2) : T2.

% Listとパターンマッチ

Γ ⊢ E1 : T, Γ ⊢ E2 : list(T)
--%------------------------------------ (T-List)
Γ ⊢ (E1::E2) : list(T).

Γ ⊢ E1 : T, Γ ⊢ E2 : list(T)
--%------------------------------------ (T-List2)
Γ ⊢ [E1|E2] : list(T).

!
--%------------------------------------ (T-Nil)
Γ ⊢ [] : list(_).

Γ ⊢ E1 : list(T1), Γ ⊢ E2 : T,
((Γ,X : T1), Y : list(T1)) ⊢ E3 : T
--%------------------------------------ (T-MatchCons)
Γ ⊢ match(E1 | [] -> E2 | X::Y->E3) : T.

% let recとエラー

(Γ,X:T1) ⊢ E1 : T1, (Γ,X:T1) ⊢ E2 : T2
--%------------------------------------ (T-LetRec)
Γ ⊢ (letrec X = E1 in E2) : T2.

!
--%------------------------------------ (T-Error)
Γ ⊢ E1 : "type error"
```

初期環境設定とMainプログラム

```
% 初期環境設定

add_env(V,E,E2):- E2=(E,V).

env -->
  add_env(print_string:(list(int)->list(int))),
  add_env(println_string:(list(int)->list(int))),
  add_env(println_int:(int->int)),
  add_env(println_bool:(bool->bool)),
  add_env(int_of_string:(list(int)->int)).

% メインプログラム

main :-
  current_prolog_flag(argv, ARGV),
  [File|_]=ARGV,
  setup_call_cleanup(open(File, read, In),
        read_string(In, _, S),
        close(In)),
  catch(term_string(E,S),error(Err,string(ErrS,ErrPos)),(write(Err),write(':'),write(ErrS),halt)),
  maplist(string_codes,ARGV,ARGV2),
  env([],Env),
  (Env,(argv:list(list(int)))) ⊢ E : T,
  (
    T="type error", write('type error\n');
    ([],(argv:ARGV2)) ⊢ E ⇓ _;
    write('runtime error\n')
  ),
  halt.
```

Prolog in ML
syntax.ml

```
type v = string * int
type t =
  | Atom of string           アトム 例) a abc hoge
  | Number of float          数値       123
  | Str of string            文字列     "abc"
  | Pred of string * t list  述語       abc(1,2) add(int(1),var("a"))
  | Var of v                 変数       A B abc(A,B)
```

```
let rec show = function
  | Atom(n)           -> n
  | Number(v)         -> string_of_float v
  | Str(v)            -> v
  | Pred(".", _) as t -> Printf.sprintf "[%s]" (show_list t)
  | Pred(n, xs)       -> Printf.sprintf "%s(%s)" n (String.concat ", " (List.map show xs))
  | Var(n, l)         -> Printf.sprintf "%s_%d" n l

and show_list = function
  | Pred(".", [t; Atom("[]")])         -> show t
  | Pred(".", [t;(Pred(".", _) as u)]) -> show t ^ show_list u
  | Pred(".", [t;u])                   -> show t ^ "|" ^ show u
  | t                                  -> show t
```

•

Prolog in OCaml
parser.mly

```
%{
open Syntax

let rec list args tail =
  match args with
  | []    -> tail
  | x::xs -> Pred(".", [x; list xs tail])
%}
%token <string> ATOM
%token <float> NUMBER
%token <string> STR
%token <string> VAR
%token <string> OP
%token LPAREN RPAREN LBRACKET RBRACKET
%token DOT OR SEMI COMMA LINE IIF
%token EOF

%right IIF
%right COMMA
%right OP

%start seq
%type <Syntax.t list> seq
%start query
%type <Syntax.t> query
%%
```

```
query:      | term DOT                  { $1 }
seq:        | sentence                  { [$1] }
            | sentence seq              { $1::$2 }
sentence:   | term DOT                  { Pred(":-", [$1; Atom "nop"]) }
            | IIF term DOT              { Pred(":-", [$2]) }
            | LINE term DOT             { Pred(":-", [$2; Atom "nop"]) }
            | term LINE term DOT        { Pred(":-", [$3; $1]) }
            | term IIF term DOT         { Pred(":-", [$1; $3]) }
term:       | term1 SEMI term           { Pred(";", [$1; $3]) }
            | term1                     { $1 }
term1:      | exp1 COMMA term1          { Pred(",", [$1; $3]) }
            | exp1 term1                { Pred(",", [$1; $2]) }
            | exp1                      { $1 }
exp1:       | exp OP exp1               { Pred($2, [$1; $3]) }
            | exp                       { $1 }
exp:        | ATOM LPAREN exps RPAREN   { Pred($1, $3) }
            | ATOM                      { Atom $1 }
            | VAR                       { Var($1, 0) }
            | NUMBER                    { Number $1 }
            | STR                       { Str $1 }
            | LBRACKET listbody RBRACKET { $2 }
            | LPAREN term RPAREN        { $2 }
exps:       | exp                       { [$1] }
            | exp COMMA exps            { $1::$3 }
listbody:   | exps                      { list $1 (Atom "[]") }
            | exps OR var_or_list       { list $1 $3 }
var_or_list: | VAR                      { Var($1, 0) }
            | LBRACKET listbody RBRACKET { $2 }
```

Prolog in OCaml
lexer.mll

```
{
  open Parser
}
let upper = ['A'-'Z'] | '\xce' ['\x91' - '\xa9']
let lower = ['a'-'z'] | '\xce' ['\xb1' - '\xbf'] |
            '\xcf' ['\x80' - '\x89']
let digit = ['0'-'9']
let atom = lower (lower|upper|digit|'_'|'`')*
let var = upper (lower|upper|digit|'_'|'`')*
let nonendl = [^'\n']*
let number = digit+ ('.' digit+)?
let str = ([^ '"' '\\'] |
           '\\' ['\\' '/' 'b' 'f' 'n' 'r' 't' '"'])*
let satom = ([^ '\'' '\\'] |
             '\\' ['\\' '/' 'b' 'f' 'n' 'r' 't' '\''])*

let op = ";" | "," | "=" | "is" | "+" | "-" | "*" | "/"
let com = [' ' '\t']* '(' [^ ')']* ')'
let ln = ('\r' '\n') | '\r' | '\n'
let ln2 = [' ' '\t']* ln [' ' '\t']*
```

```
rule token = parse
  | [' ' '\t']           { token lexbuf }
  | ln2 ln2+ '.'? ln2*   { DOT }
  | ln                   { token lexbuf }
  | ";"                  { SEMI }
  | ","                  { COMMA }
  | "("                  { LPAREN }
  | ")"                  { RPAREN }
  | "["                  { LBRACKET }
  | "]"                  { RBRACKET }
  | "." ln2*             { DOT }
  | "!"                  { ATOM("!") }
  | ":-"                 { IIF }
  | '-' '-'+ com?        { LINE }
  | op     as s          { OP s }
  | atom   as s          { ATOM s }
  | var    as s          { VAR s }
  | number as s          { NUMBER (float_of_string s) }
  | '"' (str as s) '"'   { STR (Scanf.unescaped s) }
  | "'" (satom as s) "'" { ATOM (Scanf.unescaped s) }
  | eof                  { EOF }
  | "%" nonendl          { token lexbuf }
  | _                    { token lexbuf }
```

Prolog in OCaml
prolog.ml

```
open Syntax

type e = ((string * int) * t) list    (* env *)

let rec deref e t = match t with
  | Pred (n, ts) -> Pred(n, List.map (deref e) ts)
  | Var v        -> (try deref e (List.assoc v e) with _ -> t)
  | t            -> t

let show e =
  String.concat "\n" (List.fold_right (fun ((n, l), t) ls ->
    if l >= 1 then ls else (n ^ "=" ^ Syntax.show (deref e t)) :: ls
  ) e [] )

type r = e option   (* result *)
```

Prolog in OCaml
prolog.ml

```
let rec unify e t t2 =
  let rec unify r (t, t2) = match r with
    | None -> None
    | Some e ->
      let rec bind t v t2 =
        try match List.assoc v e with
        | Var v as t3 -> if t == t3 then None else bind t v t2
        | t3          -> if t2== t3 then r    else mgu (t3, t2)
        with _ -> Some((v, t2) :: e)
      and mgu (t, t2) = match (t, t2) with
        |         t ,      Var v2  -> bind t2 v2 t
        |     Var v ,           t2  -> bind t v t2
        | Pred(x, g), Pred(x2, g2) -> if x <> x2 then None else
                                      (try List.fold_left unify r (List.combine g g2)
                                      with _ -> None)
        |         t ,           t2  -> if t = t2 then r else None
      in mgu (t, t2)
  in unify (Some e) (t, t2)

type g = t list             (* goals *)
type d = t array            (* database *)
type i = int                (* index *)
type s = (g * e * i * i) list (* stack *)
type m = g * d * i * s      (* gdis machine *)

type ('a, 'b) res = Fail of 'a | Succ of 'b
```

Prolog in OCaml
prolog.ml

```
type ('a, 'b) res = Fail of 'a | Succ of 'b
let trace = ref false

let e s = match s with
  | [] -> []
  | (_, e, _, _)::_ -> e

let el1 s = match s with
  | [] -> [],1
  | (_, e, l, _)::_ -> e,l+1

let pop m = match m with
  |   _,  d, _, ( g, _,_, i)::s -> Succ (g, d, i, s)
  |   _,  d, _,              [] -> Fail d

let uni m s t t2 =
  match unify (e s) t t2, m with
  | Some e, (_::g, d, _, (sg, _,l, i)::s) -> Succ (g, d, -1, (sg, e, l, i) :: s)
  |      _,                           m -> pop m

let rec eval e = function
  | Number i -> i
  | Pred("+", [x;y]) -> (eval e x) +. (eval e y)
  | Pred("*", [x;y]) -> (eval e x) *. (eval e y)
  | Pred("-", [x;y]) -> (eval e x) -. (eval e y)
  | Pred("/", [x;y]) -> (eval e x) /. (eval e y)
  | t -> failwith ("unknown term " ^ Syntax.show t)

let write1 e t = Printf.printf "%s%!" (Syntax.show (deref e t))
```

Prolog in OCaml
prolog.ml

```
let rec assert1 d = function
  | Pred(":-", [t]) -> process d t
  | t               -> Array.append d [| t |]

and consult1 d t =
  let filename = Syntax.show t in
  if !trace then Printf.printf "Loading %s\n" filename;
  let inp = open_in filename in
  let seq = Parser.seq Lexer.token (Lexing.from_channel inp) in
  List.fold_left assert1 d seq
```

```
and step = function
  | Fail d     -> Fail d
  | Succ (g,d,i,s as m) ->
    if !trace then Printf.printf "i=%d g=[%s],e=[%s],s=%d\n"
      i (String.concat "; " (List.map Syntax.show g)) (show (e s)) (List.length s);
    match m with
    |                      [], d,  i, s -> Succ m
    |                       g, d, -2, s -> Fail d
    | Atom "halt"          ::g, d, -1, s -> exit 0
    | Atom "nop"           ::g, d, -1, s -> step (Succ(g,d,-1,s))
    | Atom "!"             ::g, d, -1, (g2,e,l,_)::s -> step (Succ(g, d, -1, (g2, e,l, -2)::s))
    | Pred(",",  [u;v])    ::g, d, -1, s -> step (Succ(u::v::g, d, -1, s))
    | Pred(";",  [u;v])    ::g, d, -1, s -> let e,l1=el1 s in step (Succ(   u::g, d, -1, (v::g, e,l1, -1)::s))
    | Pred("=",  [u;v])    ::g, d, -1, s -> step (uni m s u v)
    | Pred("is", [u;v])    ::g, d, -1, s -> step (uni m s u (Number(eval (e s) (deref (e s) v))))
    | Pred("assert",  [t])::g, d, -1, s -> step (Succ(g, assert1 d (deref (e s) t), i, s))
    | Pred("write",   [t])::g, d, -1, s -> write1 (e s) t; step (Succ(g,d,-1,s))
    | Pred("consult", [t])::g, d, -1, s -> step (Succ(g, consult1 d (deref (e s) t), i, s))
    |                       g, d, -1, s -> step (Succ(g, d, 0, s))
```

Prolog in OCaml
prolog.ml

```
|                    t::g, d,  i, s ->
  if i >= Array.length d then step (pop m) else
  match d.(i) with
  | Pred(":-", [t2; t3]) ->
    let e,l1 = el1 s in
    let rec gen_t = function
      | Pred(n, ts) -> Pred(n, List.map (fun a -> gen_t a) ts)
      | Var(n, _)   -> Var(n, l1)
      | t -> t
    in
    begin match unify e t (gen_t t2) with
      | None   -> step (Succ(       t::g, d, i + 1, s))
      | Some e -> step (Succ(gen_t t3::g, d,    -1, (t::g, e, l1, i+1) :: s))
    end
  | t -> Printf.printf "Database is broken. %s\n" (Syntax.show t); Fail d
and solve m =
  step (match m with
    | [], _, _, _ -> pop m
    | _           -> Succ m
  )

and process d t =
  let rec prove m = match solve m with
    | Fail d -> Printf.printf "No.\n"; d
    | Succ (g, d, i, s as m) ->
      Printf.printf "%s\n" (show (e s));
      if s = [] || i = -2 then (Printf.printf "Yes.\n"; d) else (
        Printf.printf "More y/n";
        if "y" = read_line () then prove m else d
      )
  in prove ([t], d, -1, [[],[],1,-2])
```

Prolog in OCaml
main.ml

```
open Syntax
open Prolog

let welcome = "Beautiful Japanese Prolog Interpreter"

let parse str =
  Parser.query Lexer.token (Lexing.from_string str)

let help () =
  List.iter(fun (k,v) -> Printf.printf "%s\t%s\n%!" k v)
  ["q","quit"; "l","list"; "h","help";]

let rec repl d =
    Printf.printf("? %!");
    match read_line () with
    | "q"  -> ()
    | "l"  -> Array.iter (fun t ->
                Printf.printf "%s.\n%!" (Syntax.show t)
              ) d;
              repl d
    | "h"  -> help (); repl d
    | "t"  -> trace := not !trace;
              Printf.printf "Tracing %s.\n%!"
                (if !trace then "on" else "off");
              repl d
    | line -> try repl (process d (parse line))
              with Parsing.Parse_error ->
                Printf.printf "Syntax error\n%!";
                repl d
```

```
let () =
  let db = ref (consult1 [||]
                 (Atom "lib/initial.pl")) in (* load files *)
  Arg.parse
    ["-t", Arg.Set trace, "trace";]
    (fun x -> db := consult1 !db (Atom x))
    "Usage: bpij [-t] filename1 filename2 ...";
  Printf.printf "%s\n%!" (String.make (String.length welcome) '-');
  Printf.printf "%s\n%!" welcome;
  Printf.printf "%s\n%!" (String.make (String.length welcome) '-');
  help ();
  repl !db
```

Prolog in Prolog
gdis.pl

```
#!/usr/bin/env swipl --toplevel=halt --stand_alone=true -q
:- initialization(main).
:- style_check(-singleton).

% prolog

pop((_, D, _, [(G, I)|S]), succ(G, D, I, S)).
pop((_, D, _,         []), fail(D)).

uni(M,T,T,M2) :- M=([_|G], D, _, S), M2=succ(G, D, D, S).
uni(M,_,_,M2) :- pop(M, M2).

assert1(':-'(T),D, R) :- process(D, [], T, R).
assert1(T      ,D, R) :- append(D, [T], R).

read_stream_to_terms(Stream, RC, REnv) :-
  read_term(Stream, C, [variable_names(Env)]), !,
  ( C = end_of_file, RC = [], REnv = [],!
  ; read_stream_to_terms(Stream, RC1, REnv1),
    RC = [C | RC1], append(Env, REnv1, REnv), !).

cnv2(A :- B, A :- B).
cnv2(:- B, :- B).
cnv2(T, T :- nop).

consult1(Filename,D,D2) :-
  (flag(trace,1,1),write("Loading "),write(Filename),nl;!),!,
  setup_call_cleanup(open(Filename, read, In), read_stream_to_terms(In, Terms, Env), close(In)),
  maplist(cnv2,Terms,Terms2),!,
  foldl(assert1, Terms2, D, D2),!.
```

Prolog in Prolog
gdis.pl

```
step(succ(G,D,I,S), R2) :- (flag(trace,1,1),format("i=~p G=~p S=~p~n",[I,G,S]);!),!, step1((G,D,I,S), G,D,I,S, R2).
step(R, R).

step1(M,   [ ], D, I,         S, R) :- R=succ([], D,  I, S),!.
step1(M, [T|G], D, [T1=T2|_], S, R) :- copy_term(T1,T),call(T2,M,M,R).
step1(M, [T|G], D, [T_|I1],   S, R) :- copy_term(T_,':-'(T,T3)), step(succ([T3|G], D, D, [([T|G], I1)|S]),R).
step1(M, [T|G], D, [T_|I1],   S, R) :- step(succ([T |G], D, I1, S), R).
step1(M,    G , D, I,         S, R) :- pop(M,M2), step(M2,R).

solve(M, R2) :- ([], _, _, _) = M, pop(M, R), step(R, R2).
solve(M, R2) :- ( G, D, I, S) = M, step(succ(G,D,I,S), R2).
```

```
read_line(P, A) :-
  prompt1(P), read_line_to_codes(user_input,R),atom_codes(A,R).

env_show_add_(K=V,R) :- format(atom(R),"~s=~p",[K,V]).
env_show(Vs,R) :- maplist(env_show_add_,Vs,E3), atomic_list_concat(E3,', ',R).

prove(Env,M,D2) :- solve(M, R),
  ( R = fail(D2)
  ; R = succ(G, D, I, S),
    env_show(Env, X), write(X),nl,
    ( (S=[]; S=[(fail,D)]), write("Yes.\n"), D2=D
    ; ( read_line("More y/n", "y"), prove(Vs,(G, D, I, S), D2)
      ; D2=D))).

process(D, Env,T, D2) :- prove(Env,([T], D, D, [([fail],D)]), D2).
```

Prolog in Prolog
gdis.pl

```
gen_builtin(T1 :- T2, T = N) :- T1 =.. [N,_,([T|_],_,_,_),_], assert(T1 :- T2).

init_db(Db) :- maplist(gen_builtin,[
    builtin_halt(   M,([      halt|G], D, _, S), R) :- halt,
    builtin_fail(   M,([      fail|G], D, _, S), R) :- R=fail(D),
    builtin_nop(    M,([       nop|G], D, _, S), R) :- step(succ(G, D, D, S), R),
    builtin_cut(    M,([         !|G], D, _, S), R) :- step(succ(G, D, D, [([fail], D)]), R),
    builtin_comma(  M,([(U,V)     |G], D, _, S), R) :- step(succ([U,V|G], D, D, S), R),
    builtin_semi(   M,([(U;V)     |G], D, _, S), R) :- step(succ([U|G], D, D, [([V|G], D)|S]), R),
    builtin_eq(     M,([(U=V)     |G], D, _, S), R) :- (uni(M,U,V,R1),step(R1, R)),
    builtin_is(     M,([(U is V)  |G], D, _, S), R) :- (N is V, !, uni(M,U,N,R1), step(R1,R)),
    builtin_write(  M,([  write(T)|G], D, _, S), R) :- (write(T), step(succ(G, D, D, S), R)),
    builtin_assert( M,([ assert(T)|G], D, _, S), R) :- (assert1(T, D, D2), step(succ(G, D2, D, S), R)),
    builtin_consult(M,([consult(T)|G], D, _, S), R) :- (consult1(T, D, D2), step(succ(G, D2, D, S),R))
], Db).
```

Prolog in Prolog
gdis.pl

```
welcome("Beautiful Japanese Prolog Interpreter").

help :- maplist(format('~s\t~s\n'),[["e","exit"], ["l","list"],["h","help"]]).

syntax_print(T) :- write(T),write('.'),nl.

repl(D) :-
    read_line("? ", Y),!,
    (Y='e'
    ;Y='l', maplist(syntax_print, D),!, repl(D)
    ;Y='h', help,!, repl(D)
    ;Y='t', flag(trace,T,1-T),(T=0,S='on';S='off'),format('Tracing ~s~n',[S]),!,repl(D)
    ;       term_string(R,Y,[variable_names(Env)]),!,process(D,Env, R, D2),!, repl(D2)
    ;       write('Syntax error\n'),!, repl(D)).optParse([         ],Db,Db ).

optParse(['-t'|Args],Db,Db2) :- flag(trace,_,1), optParse(Args,Db,Db2).
optParse([   A|Args],Db,_  ) :- sub_atom(A,0,1,_,'-'), write("Usage: bjpl [-t] filename1 filename2 ...\n"), halt.
optParse([   A|Args],Db,Db2) :- consult1(A,Db, Db1), optParse(Args, Db1, Db2).

main :-
  current_prolog_flag(argv, ARGV),
  flag(trace,_,0),
  init_db(Init_db),
  consult1('lib/initial.pl',Init_db, Db), % load files
  optParse(ARGV, Db, Db1),!,
  welcome(W),atom_length(W,L),
  (between(1, L, _), write(-), fail;nl),
  write(W),nl,
  (between(1, L, _), write(-), fail;nl),
  help,
  repl(Db1).halt.
```