ocaml 第五期レポート課題

高口 奨一郎
学籍番号：6319045

2020 年 8 月 31 日

1　解いた問題

問題 1,2,3,6,4,5

2　問題 1

SProlog の生成規則と抽象構文木の対応を参考にしながら，SProlog 処理系を完成させなさい

2.1　プログラムソース

```
module Lexer = struct

type token = CID of string | VID of string | NUM of string
            | TO | IS | QUIT | OPEN | EOF | ONE of char

module P = Printf

exception End_of_system

let _ISTREAM = ref stdin

let ch = ref []

let read () = match !ch with [] -> input_char !_ISTREAM
                           | h::rest -> (ch := rest; h)

let unread c = ch := c::!ch

let lookahead () = try let c = read () in unread c; c with End_of_file -> '$'

let rec integer i =
(* 文字列として数字を構成 *)
 let c = lookahead () in
    if (c >= '0' && c <= '9') then
          integer (i^(Char.escaped (read ())))
    else i

and identifier id =
```

```
 let c = lookahead () in
    if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') ||
       (c >= '0' && c <= '9') || c == '_') then
          identifier (id^(Char.escaped (read ())))
    else id

and native_token () =
 let c = lookahead () in
   if (* CID に対する識別子および予約語 *)
      (c >= 'a' && c <= 'z') then
         let id = identifier "" in
             match id with
              "is" -> IS
             |"quit" -> QUIT
             |"open" -> OPEN
             |"eof" -> EOF
             |_ -> CID (id)
   else if (* VID に対する識別子 *)
      (c >= 'A' && c <= 'Z') then VID (identifier "")
   else if (c >= '0' && c <= '9') then NUM (integer "")
   else if (* :- を認識して TO を返す *)
      (c= ':') then  (let sub1 = read() in let sub2 = read() in
                       if sub2 = '-' then TO
                       else ONE (sub1))
   else ONE (read ())

and gettoken () =
try
let token = native_token () in
  match token with
     ONE ' ' -> gettoken ()
   | ONE '\t' -> gettoken ()
   | ONE '\n' -> gettoken ()
   | _ -> token
with End_of_file -> EOF

let print_token tk =
match tk with
(CID i) -> P.printf "CID(%s)" i
| (VID i) -> P.printf "VID(%s)" i
```

```
  | (NUM i) -> P.printf "NUM(%s)" i
  | (TO) -> P.printf ":-"
  | (QUIT) -> P.printf "quit"
  | (OPEN) -> P.printf "open"
  | (IS) -> P.printf "is"
  | (EOF) -> P.printf "eof"
  | (ONE c) -> P.printf "ONE(%c)" c

let rec run () =
   flush stdout;
   let rlt = gettoken () in
       match rlt with
         (ONE '$') -> raise End_of_system
        |_ -> (print_token rlt; P.printf "\n"; run())

end

module Evaluator = struct
(* 抽象構文木の型宣言 *)
type ast = Atom of string | Var of string | App of
  string * ast list

(* 抽象構文木の印字関数 *)
module P = Printf
let rec print_ast ast = match ast with
   (App(s, hd::tl)) -> (P.printf "App(\"%s\",[" s;
      print_ast hd; List.iter (fun x -> (print_string
          ";"; print_ast x)) tl;
         print_string "])")
   | (App(s, [])) -> P.printf "App(\"%s\",[])" s
   | (Atom s) -> P.printf "Atom \"%s\"" s
   | (Var s) -> P.printf "Var \"%s\"" s
let print_ast_list lst = match lst with
   (hd::tl) -> (print_string "["; print_ast hd;
      List.iter (fun x -> (print_string ";";
      print_ast x)) tl; print_string "]")
   | [] -> print_string "[]"

(* 関数 sub, mgu, succeed, rename solve eval の定義 *)
let sub name term =
   let rec mapVar ast = match ast with
      (Atom x) -> Atom(x)
    | (Var n) -> if n=name then term else Var n
```

```
        | (App(n, terms)) -> App(n, List.map mapVar terms)
in mapVar

let mgu (a,b) =
   let rec ut (one, another, unifier) = match (one, another) with
       ([], []) -> (true, unifier)
     | (term::t1, Var(name)::t2) ->
       let r = fun x -> sub name term (unifier x) in
          ut(List.map r t1, List.map r t2, r)
     | (Var(name)::t1, term::t2) ->
       let r = fun x -> sub name term (unifier x) in
          ut(List.map r t1, List.map r t2, r)
     | (Atom(n)::t1, Atom(m)::t2) ->
       if n=m then ut(t1,t2,unifier) else (false, unifier)
     | (App(n1,xt1)::t1, App(n2,xt2)::t2) ->
       if n1=n2 && List.length xt1 = List.length xt2 then
          ut(xt1@t1, xt2@t2, unifier)
       else (false, unifier)
     | (_,_) -> (false, unifier);
   in ut ([a],[b], (fun x -> x))

let succeed query = (print_ast query; true)

let rename ver term =
  let rec mapVar ast = match ast with
        (Atom x) -> Atom(x)
      | (Var n) -> Var(n^"#"^ver)
      | (App(n, terms)) -> App(n, List.map mapVar terms)
  in mapVar term

exception Compiler_error

let rec solve (program, question, result, depth) = match question with
  [] -> succeed result
 | goal::goals ->
   let onestep _ clause =
    match List.map (rename (string_of_int depth))
```

```
        clause with
        [] -> raise Compiler_error
      | head::conds ->
        let (unifiable, unifier) = mgu(head,goal) in
          if unifiable then
            solve (program, List.map unifier (conds@goals),
                                    unifier result, depth+1)
          else true
    in List.fold_left onestep true program

let eval (program, question) = solve(program, [question], question, 1)

end

module Parser = struct

module L = Lexer

module E = Evaluator

let tok = ref (L.ONE ' ')

let getToken () = L.gettoken ()

let advance () = (tok := getToken())

exception Syntax_error

let error () = raise Syntax_error

let check t = match !tok with
      L.CID _ -> if (t = (L.CID "")) then () else error ()
    | L.VID _ -> if (t = (L.VID "")) then () else error()
    | L.NUM _ -> if (t = (L.NUM "")) then () else error()
    | tk -> if (tk=t) then () else error()

let eat t = (check t; advance())

let prog = ref [[E.Var ""]]
```

```
let rec clauses() = match !tok with
    L.EOF -> []
   | _ -> let a = clause() in let b = clauses() in a::b

and clause() = match !tok with
    L.ONE '(' -> let a = [term()] in eat(L.ONE '.') ; a
   | _ -> let a = predicate() in let b = to_opt() in eat(L.ONE '.') ; a::b

and to_opt() = match !tok with
    L.TO -> eat(L.TO) ; let b = terms() in b
   | _ -> []

and command() = match !tok with
    L.QUIT -> exit 0
   | L.OPEN -> (eat(L.OPEN);
      match !tok with
       L.CID s -> (eat(L.CID ""); check (L.ONE '.');
       L._ISTREAM := open_in (s^".pl"); advance();
          prog := clauses(); close_in (!L._ISTREAM))
   |_ -> error())
   |_ -> let t = term() in
          (check(L.ONE '.'); let _ = E.eval(!prog, t) in ())

and term() = match !tok with
    L.ONE '(' -> (eat(L.ONE '(') ; let b = term() in eat(L.ONE ')') ; b)
   |L.VID s -> (eat(L.VID "") ; eat(L.IS) ; expr())
   | _ -> let a = predicate() in a

and terms() = (let a = [term()] in let c = terms' () in a @ c)

and terms' () = match !tok with
    L.ONE ',' -> (eat(L.ONE ',') ; let d = [term()] in let e = terms'() in d @ e)
   | _ -> []

and predicate() = match !tok with
```

```
    L.CID a -> (eat(L.CID "") ; eat(L.ONE '(') ; let b = args() in eat(L.ONE ')') ; E.App (a,b))
  | _ -> error()

and args() = let a =[expr()] in let c = args'() in a @ c

and args'() = match !tok with
    L.ONE ',' -> (eat(L.ONE ',') ; let d = [expr()] in let e = args'() in d @ e\
)
  | _ -> []

and expr() = match !tok with
    L.ONE '(' -> (eat(L.ONE '(') ; let b =  expr() in eat(L.ONE ')') ; b)
  | L.ONE '[' -> (eat(L.ONE '[') ; let a = list() in eat(L.ONE ']') ; a)
  | L.CID s -> (eat(L.CID "") ; tail_opt s)
  | L.VID s -> (eat(L.VID "") ; E.Var s)
  | L.NUM n -> (eat(L.NUM "") ; E.Atom n)
  | _ -> error()

and tail_opt s = match !tok with
     L.ONE '(' -> (eat(L.ONE '(') ; let a = args() in eat(L.ONE ')') ; E.App (s\
 , a))
  | _ -> E.Atom s

and list() = match !tok with
    L.ONE ']' -> E.Atom "nil"
  | _ -> E.App ("cons" , [expr() ; list_opt()])

and list_opt() = match !tok with
    L.ONE '|' -> (eat(L.ONE '|')) ; id()
  | L.ONE ',' -> (eat(L.ONE ',')) ; list()
  | _ -> E.Atom "nil"

and id() = match !tok with
    L.CID a -> (eat(L.CID "")) ; E.Atom a
  |L.VID a -> (eat(L.VID "")) ; E.Var a
  |L.NUM a -> (eat(L.NUM "")) ; E.Atom a
  | _ -> error()
```

```

end

let rec run() =
   print_string "?- ";
   while true do
     flush stdout; Lexer._ISTREAM := stdin;
     Parser.advance(); Parser.command() ;
       print_string "\n?-"
done

let _ = run()
```

2.2　左再帰を除いた文法

新しく terms'、args' を定義することで左再帰を除き、右再帰に変換することができる。

2.2.1　terms

terms → term terms'

terms' → "," term terms'

terms' →

2.2.2　args

args → expr args'

args' → "," expr args'

args' →

2.3　プログラムの説明

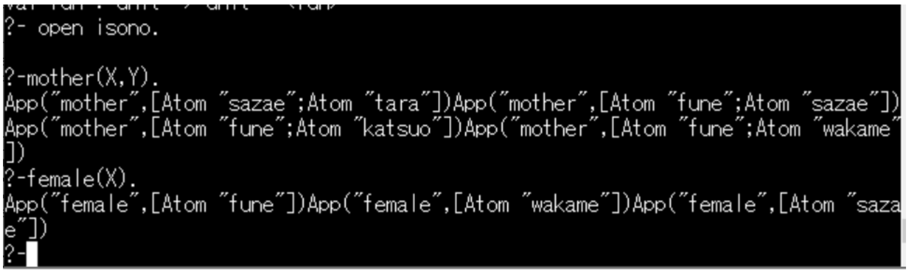
第四期の課題で作成した Lexer と Parser を用いて、Parser の中身を与えられた文法に従って構成し直した。前回と同様に右再帰の関数は上記のように文法を作り直した。また、Evaluator に関しては授業プリントを参考にしながら作成した。

3　問題 2

isono プログラムを入力し，いくつかの質問について，振舞いを確認しなさい．

3.1　実行結果

以下のように正しく動作していることがわかる。



4　問題 3

第 9 章で示した関数 succeed では，インスタンス化した質問の構文木を印字するようになっている．これを，SProlog のソース言語の表現で印字するようにしなさい

4.1　プログラムソース

Evaluator 内の抽象構文木について以下のような変更を加えた。

```
module P = Printf
let rec print_ast ast = match ast with
    (App(s, hd::tl)) -> (P.printf "%s(" s;
        print_ast hd; List.iter (fun x -> (print_string ";"; pri
            print_string ")")
    | (App(s, [])) -> P.printf "%s" s
    | (Atom s) -> P.printf "%s" s
    | (Var s) -> P.printf "%s" s
let print_ast_list lst = match lst with
```

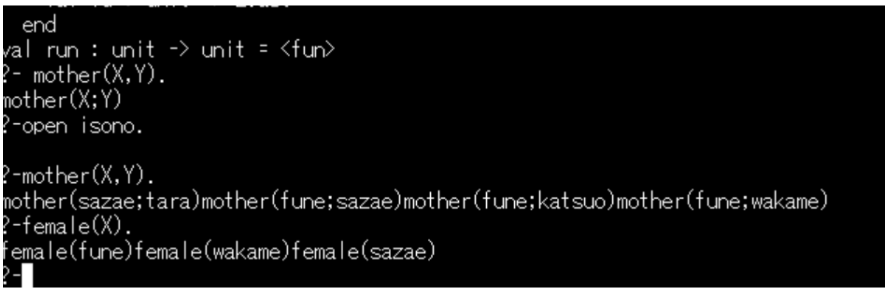
```
    (hd::tl) -> (print_string "["; print_ast hd;
        List.iter (fun x -> (print_string ";";
        print_ast x)) tl; print_string "]")
    | [] -> print_string "[]"
```

4.2 プログラムの説明

抽象構文木の印字関数の部分を Sprolog のソース言語の表現と同じになるように変更を加えた。

4.3 実行結果

以下のように正しく動作していることがわかる。



```
  end
val run : unit -> unit = <fun>
?- mother(X,Y).
mother(X;Y)
?-open isono.

?-mother(X,Y).
mother(sazae;tara)mother(fune;sazae)mother(fune;katsuo)mother(fune;wakame)
?-female(X).
female(fune)female(wakame)female(sazae)
?-
```

5 問題 6

[,」で並べた複数質問を受け付けられるように拡張せよ

5.1 プログラムソース

```
let rec print a b =
  match (a, b) with
     [],[] -> ()
    |[],hd :: tl -> ()
    |hd :: tl,[] -> ()
    |(hd1 :: tl1 , hd2 :: tl2)-> (print_ast hd1 ;
       print_string "=" ;print_ast hd2;print_str\
ing "\n" ;print tl1 tl2)

let rec solve (program, question, result, a, b, depth
   ) =
```

```
 match question with
   [] -> succeed result a b
 | goal::goals ->
   let onestep _ clause =
    match List.map (rename (string_of_int depth)) clause with
       [] -> raise Compiler_error
     | head::conds ->
       let (unifiable, unifier) = mgu(head,goal) in
        if unifiable then
          solve (program, List.map unifier (conds@goals),
                             List.map unifier result, a,
                             List.map unifier b, depth + 1)
        else true
in List.fold_left onestep true program

let rec get x =
   match x with
     [] -> []
    |hd :: tl -> match hd with
                    Var t -> Var t :: (get tl)
                   |Atom s -> get tl
                   |App(x,y) -> (get y) @ (get tl)

let rec find y =
   let rec sub n lst =
        match lst with
          [] -> []
         |hd :: tl -> if hd = n then sub n tl
                      else hd :: sub n tl
   in match y with
        [] -> []
       |hd :: tl -> hd :: (find (sub hd y))

let eval (program, question) =
let l = find (get question)
in solve (program,question,question,l,l,1)

and command() = match !tok with
    L.QUIT -> exit 0
   | L.OPEN -> (eat(L.OPEN);
      match !tok with
```

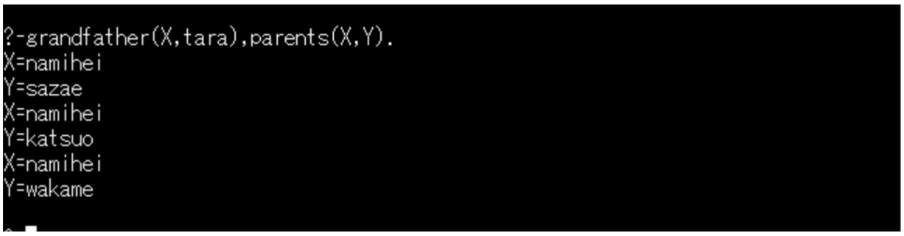
```
        L.CID s -> (eat(L.CID ""); check (L.ONE '.');
        L.ISTREAM := open_in (s^".pl"); advance();
          prog := clauses(); close_in (!L.ISTREAM))
    |_ -> error())
    |_ -> let t = terms() in
          (check(L.ONE '.'); let _ = E.eval(!prog, t)
             in ())
```

5.2 プログラムの説明

抽象構文木の部分と command 関数に変更を加えることで複数質問に答えられるようにした。

5.3 実行結果

以下のように正しく動作していることがわかる。



6 問題 4、問題 5

通常の Prolog は，質問が真であったとき，インスタンス化した質問を印字するのではなく，質問に含まれる変数ごとに，対応する項を印字する．SProlog の処理系もそのように拡張せよ

通常の Prolog は, 質問が真である単一化代入が得られるたびに, 推論を継続するか終了するかを指示することができる．SProlog の処理系も, succeed が呼ばれるたびに, 実行を中断し, [;」を打ち込むと継続, [.」を打ち込むと, それ以降の処理を回避するように拡張しなさい

6.1 プログラムソース

```
let succeed query a b =
(print a b ;print_string "Yes";flush stdout;
```

```
let y = ref ' ' in while (!y != ';'&& !y != '.') do
y := input_char stdin done;flush stdout;
if !y = ';' then true else raise Error)
let rec run() =
   print_string "?- ";
   while true do
     flush stdout; Lexer._ISTREAM := stdin;
     Parser.advance();(try Parser.command() with
        Evaluator.Error ->
print_string "No"); print_string "\n?-"
```

6.2 プログラムの説明

問題 6 から印字関数に手を加えることで表記を正しいものに変更した。

6.3 実行結果

以下のように正しく動作していることがわかる。