Exercise 4

鵜飼昌樹

平成 20 年 6 月 6 日

Exercise 4

4.1 正規表現の抽象文法を表現する ML datatype を書け

構文要素は * () | ? . としよう。Syntax Sugar (+ や [] 等) は抽象構文では消えるので無視。ϵ の導入によって ? も消せるか。また () は抽象文法で消えるし、| も regexp の list と 扱うとしたら、以下の通り。

```
datatype
 regexp = SimpleChar of char
        | ModifyChar of char * meta    // これ不要でもいいな
        | ModifyStr  of regexp * meta
        | RegexpStr  of regexp list
        | RegexpOr   of regexp list    // ( | | ... )
and
 char   = Char    of int
        | Empty
and
 meta   = ZeroMore | AnyChr
```

4.2 評価伝播不良の修正

Program 4.4 で straight-line プログラムを翻訳すると、式の内部に組み込まれた文が恒久的な効果を持っていない：任意の代入文はカッコを閉じると “隠蔽” されてしまう。つまり、このプログラム

```
 a := 6; a := (a := a+1, a+4) + a; print (a)
```

は 18 ではなく 17 を表示する。この問題を修正するために、 exp の意味値の型を変更し、値と新しい表を生成するようにせよ; このように

```
%nonterm exp of table -> (table*int)
       | stm of table -> table
       | exps of table -> table
       | prog of table
```

これに応じて Program 4.4 のセマンティックアクションも変更せよ。

... ところが掲示のプログラムそのままでは動いてくれないのよ。/ が div(整数除算) でないあたり、最初から動かない前提のような気もするが。ともかく解答、yacc 部分のみ。

```
type table = string -> int
fun update (t,id,(t2,num)) = fn s => if s=id then num else t s
val emptytable = fn j => raise Fail ("uninitialized var: " ^ j)
%%
%term
      INT of int
    | ID of string
    | PLUS | MINUS | TIMES | DIV | ASSIGN | PRINT
    | LPAREN | RPAREN | COMMA | SEMICOLON | EOF
%nonterm
         exp of table -> (table * int)
       | stm of table -> table
       | exps of table -> table
       | prog of table
%right SEMICOLON
%left PLUS MINUS
%left TIMES DIV

%verbose
%start prog
%eop EOF
%pos int
%noshift EOF
%name Tiger          (* This is necessary *)
%value ID ("hoge")
%value INT (999)
%%
prog: stm                      (stm(emptytable))

stm: stm SEMICOLON stm         (fn t:table => stm2 (stm1 (t) ) )
   | ID ASSIGN exp             (fn t:table => update (t,ID1,exp1 t) )
   | PRINT LPAREN exps RPAREN (fn t:table => (exps1 t; t))

exps: exp             (fn t => let val (t2,n) = exp t in
                                   print (Int.toString n); t2 end)
    | exps COMMA exp (fn t => let val (t2,n) = exp t in
                                   print (Int.toString n); exps t2 end)
exp: INT           (fn t => (t,INT) )
   | ID            (fn t => (t,t(ID))) (* lookup *)
```

```
  | exp PLUS exp  (fn t =>
                     let val (e1t,e1n) = exp1(t) val (e2t,e2n) = exp2(e1t) in
                         (e2t, e1n + e2n) end)
  | exp MINUS exp (fn t =>
                     let val (e1t,e1n) = exp1(t) val (e2t,e2n) = exp2(e1t) in
                         (e2t, e1n - e2n) end)
  | exp TIMES exp (fn t =>
                     let val (e1t,e1n) = exp1(t) val (e2t,e2n) = exp2(e1t) in
                         (e2t, e1n * e2n) end)
  | exp DIV exp   (fn t =>
                     let val (e1t,e1n) = exp1(t) val (e2t,e2n) = exp2(e1t) in
                         (e2t, e1n div e2n) end)
  | stm COMMA exp (fn t => exp(stm(t)))
  | LPAREN exp RPAREN ( exp )
```

4.3 純粋関数型

Program 4.4 は純粋関数型言語ではない; PRINT のセマンティックアクションには、副作用のある ML の print 文が含まれる。純粋関数型のインタプリタを作ることは出来る。各文において、表だけでなく、print したかどうかの値を持つリストを返すようにすれば良い。つまり:

```
%nonterm stm of table -> (table * int list )
       | prog of int list
```

これに応じて exp や exps の型を補正して Program 4.4 を書き直せ。

正格言語であるところの SML でこれを導入しても変化が無いから対処のしかたが正しいのかどうかがわからんという。そもそも問題文の訳が正しいのかどうかすら怪しい。int list には何を返すのが正しいのか?

「print したかどうかの値を持つ」という解釈で正しいのなら、Concurrent Clean の一意型の方式かしら。「Print したい値を持つ」なら Haskell monad 方式っぽい。

本当は stm は連続するときに伝播させないといけないはずで、つまり stm の型は `table * int list -> table * int list` にしないといけないけれど、面倒なので省略。

```
type table = string -> int
fun update (t,id,(t2,num)) = fn s => if s=id then num else t s
val emptytable = fn j => raise Fail ("uninitialized var: " ^ j)
%%
%term
      INT of int
    | ID of string
    | PLUS | MINUS | TIMES | DIV | ASSIGN | PRINT
```

```
    | LPAREN | RPAREN | COMMA | SEMICOLON | EOF
%nonterm
         exp of table -> (table * int)
       | stm of table -> (table * int list)
       | exps of table -> (table * int list)  (* ? *)
       | prog of int list
%right SEMICOLON
%left PLUS MINUS
%left TIMES DIV

%verbose
%start prog
%eop EOF
%pos int
%noshift EOF
%name Tiger          (* This is necessary *)
%value ID ("hoge")
%value INT (999)
%%
prog: stm                     (let val (tbl, il) = stm(emptytable) in il end)

stm: stm SEMICOLON stm        (fn t => let val (t1,il1) = stm1 (t)
                                           val (t2,il2) = stm2 (t1) in
                                           (t2, il2 @ il1) end )
   | ID ASSIGN exp            (fn t => ( update (t,ID1,exp1 t), []) )
   | PRINT LPAREN exps RPAREN (fn t => (exps1 t; (t,[])) )

exps: exp            (fn t => let val (t2,n) = exp t in
                                  print (Int.toString n); (t2,[n])
                              end)
    | exps COMMA exp (fn t => let val (t2,n) = exp t
                                  val _  = print (Int.toString n)
                                  val (t3,n2) = exps t2
                              in
                                  (t3, n2 @ [n]) end)
exp: INT           (fn t => (t,INT) )
   | ID            (fn t => (t,t(ID))) (* lookup *)
   | exp PLUS exp  (fn t =>
                       let val (e1t,e1n) = exp1(t) val (e2t,e2n) = exp2(e1t) in
                           (e2t, e1n + e2n) end)
   | exp MINUS exp (fn t =>
                       let val (e1t,e1n) = exp1(t) val (e2t,e2n) = exp2(e1t) in
                           (e2t, e1n - e2n) end)
```

```
   | exp TIMES exp (fn t =>
                      let val (e1t,e1n) = exp1(t) val (e2t,e2n) = exp2(e1t) in
                          (e2t, e1n * e2n) end)
   | exp DIV exp   (fn t =>
                      let val (e1t,e1n) = exp1(t) val (e2t,e2n) = exp2(e1t) in
                          (e2t, e1n div e2n) end)
   | stm COMMA exp (fn t =>
                      let val (t1,il1) = stm (t)
                          val t2 = exp (t1) in (t2) end)
   | LPAREN exp RPAREN ( exp )
```

あるいは

```
type table = string -> int
fun update (t,id,(t2,num)) = fn s => if s=id then num else t s
val emptytable = fn j => raise Fail ("uninitialized var: " ^ j)
%%
%term
      INT of int
    | ID of string
    | PLUS | MINUS | TIMES | DIV | ASSIGN | PRINT
    | LPAREN | RPAREN | COMMA | SEMICOLON | EOF
%nonterm
         exp of table -> (table * int)
       | stm of table -> (table * int list)
       | exps of table -> (table * int list)  (* ? *)
       | prog of int list
%right SEMICOLON
%left PLUS MINUS
%left TIMES DIV

%verbose
%start prog
%eop EOF
%pos int
%noshift EOF
%name Tiger          (* This is necessary *)
%value ID ("hoge")
%value INT (999)
%%
prog: stm                     (let val (tbl, il) = stm(emptytable) in il end)

stm: stm SEMICOLON stm        (fn t => let val (t1,il1) = stm1 (t)
                                           val (t2,il2) = stm2 (t1) in
```

```
                                        (t2, il2 @ il1) end )
   | ID ASSIGN exp            (fn t => ( update (t,ID1,exp1 t), []) )
   | PRINT LPAREN exps RPAREN (fn t => (exps1 t; (t,[])) )

exps: exp            (fn t => let val (t2,n) = exp t in
                                  print (Int.toString n); (t2,[n])
                              end)
    | exps COMMA exp (fn t => let val (t2,n) = exp t
                                  val (t3,n2) = exps t2
                              in
                                  (t3, n2 @ [n]) end)
exp: INT           (fn t => (t,INT) )
   | ID            (fn t => (t,t(ID))) (* lookup *)
   | exp PLUS exp  (fn t =>
                       let val (e1t,e1n) = exp1(t) val (e2t,e2n) = exp2(e1t) in
                           (e2t, e1n + e2n) end)
   | exp MINUS exp (fn t =>
                       let val (e1t,e1n) = exp1(t) val (e2t,e2n) = exp2(e1t) in
                           (e2t, e1n - e2n) end)
   | exp TIMES exp (fn t =>
                       let val (e1t,e1n) = exp1(t) val (e2t,e2n) = exp2(e1t) in
                           (e2t, e1n * e2n) end)
   | exp DIV exp   (fn t =>
                       let val (e1t,e1n) = exp1(t) val (e2t,e2n) = exp2(e1t) in
                           (e2t, e1n div e2n) end)
   | stm COMMA exp (fn t =>
                       let val (t1,il1) = stm (t)
                           val t2 = exp (t1) in (t2) end)
   | LPAREN exp RPAREN ( exp )
```

後者の解釈の場合、parse.sml 側で Parse した結果の int list を受け取って表示させるようにする必要がある。

4.4 Exercise 4.2, 4.3 の考え方を結合して、純粋関数型バージョンのインタプリタを書け。表示および文を内包する式を正しく制御せよ。

ということは Exercise4.3 は Ex4.2 ベースで書くのでは無く、Program 4.4 ベースでかけということだったんだろうか。なら Ex4.3 が解答でいいですか :-)

4.5 Program 4.4 を再帰下降パーサで、パース関数内にセマンティックアクションを組み込んだ形で実装せよ。

つまり LL(1) なインタプリタを書けば良いのか。構文ルールを確認しておこう。

```
P -> S
S -> S ; S
     ID := E
     print ( F )
F -> E
     F , E
E -> INT
     ID
     E + E
     E - E
     E * E
     E / E
     S , E
     ( E )
```

なんと左再帰の多いことよ (当たり前か)。

```
0 S -> S
1 S -> ID := E T
2      print ( F ) T
3 T -> ; S            (ここで右辺の最後に T は不要だな)
4      ε
5 F -> E , F
6      E
7 E -> INT D
8      ID  D
9      S , E          (ここで右辺の最後に D は不要だな)
10     ( E ) D
11D -> + E            (ここで右辺の最後に D は不要だな)
12     - E
13     * E
14     / E
15     ε
```

つまり右辺の終端に右再帰文字があるときには X' 相当は不要と?
これで LL(1) になっているんでしょうかー。各集合を求める：

```
  first           null  follow
S ID print        no    folT , $
T ;               yes   folS
F 1stE            no    )
E INT ID ( 1stS   no    folF 1stT folD , )
D + - * /         yes   folE

  first           null  follow
```

```
S ID print        no    , $
T ;              yes   , $
F INT ID ( print  no    )
E INT ID ( print  no    ; , )
D + - * /        yes   ; , )
```

```
  ID print ; , INT ( ) + - * /   $
S  1   2
T          3 4                  4
F **  **       ** **
E 89            7  10
D          1515      15 11 .. 14
```

おおう、共通項括り出しを忘れてた。でも E -> ID D と E-> S,E がぶつかるな。LL(2) でないとだめ？E -> S,E を展開すればいいけどめんどいな。

```
 1  S -> S1
 2       S2
 3  S1-> ID := E T
 4  S2-> print ( F ) T
 5  T -> ; S
 6       ε
 7  F -> E G
 8  G -> , F
 9       ε
 0  E -> INT D
11       ID E'      (もとの S1 と E の分岐)
12       S2 , E
13       ( E ) D
14  E'-> D             (ここ E の部分)
15       := E T , E    (ここ S1 の展開)
16  D -> + E
17       - E
18       * E
19       / E
20       ε
```

```
  ID print ; , INT ( ) + - * / := $
S  1   2
S1 3
S2     4
T          5 6                   6
F 7   7         7  7
G            8       9
```

```
E 11  12        10 13
E'          1414      141414141415
D           2020      2016171819
```

こうかな?(ちゃんと集合を求め直すべきですね！)。
SML のライブラリを調べるのが面倒だった (特にレキサ関連の作成が) ので、OCaml で書いたよ。

```
(* Chapter4 Exercise 4.5 : めどいので OCaml で書いたよ! m.ukai *)

#load "str.cma" (*正規表現ライブラリのロード *)

type token =
  | ID  of string
  | INT of int
  | MUL | DIV | MINUS | PLUS
  | ASSIGN | LPAR | RPAR
  | PRINT | COMMA | SEMI
  | EOF

let variable = Hashtbl.create 10

let eat stream token = match stream with
  | [] ->  exit 0   (* おしまい *)
  | hd :: tl -> if hd = token then tl else raise Not_found
let lookahead stream = match stream with [] -> EOF | hd::tl -> hd

let parse2stream str =
  let eop = String.length str in
  let token_print = Str.regexp "print[^0-9a-zA-Z_]" in
  let token_id = Str.regexp "[a-zA-Z_][a-zA-Z_0-9]*" in
  let token_int = Str.regexp "[0-9]+" in
  let token_skip = Str.regexp "[ \t\n]" in
  let token_ass = Str.regexp ":=" in
  let rec p2s ptr plist =
    if ptr >= eop then plist else
      if Str.string_match token_print str ptr then p2s (ptr+5) (PRINT::plist)
      else if Str.string_match token_skip str ptr then p2s (ptr+1) plist
      else if Str.string_match token_ass str ptr then p2s(ptr+2)(ASSIGN::plist)
      else if Str.string_match token_id str ptr then
        let idstr = Str.matched_string str in
          p2s (ptr+(String.length idstr)) ((ID idstr)::plist)
      else if Str.string_match token_int str ptr then
        let intstr = Str.matched_string str in
          p2s (ptr+(String.length intstr)) ((INT(int_of_string intstr))::plist)
```

```
      else if str.[ptr] = '+' then p2s (ptr+1) (PLUS::plist)
      else if str.[ptr] = '-' then p2s (ptr+1) (MINUS::plist)
      else if str.[ptr] = '*' then p2s (ptr+1) (MUL::plist)
      else if str.[ptr] = '/' then p2s (ptr+1) (DIV::plist)
      else if str.[ptr] = ',' then p2s (ptr+1) (COMMA::plist)
      else if str.[ptr] = ';' then p2s (ptr+1) (SEMI::plist)
      else if str.[ptr] = '(' then p2s (ptr+1) (LPAR::plist)
      else if str.[ptr] = ')' then p2s (ptr+1) (RPAR::plist)
      else raise Not_found
  in
    List.rev (p2s 0 [])

exception Empty_sem

let rec parse_prog stream =
  parse_stm stream
and parse_stm stream =
  match lookahead stream with
    | ID s  -> parse_stm1 stream
    | PRINT -> parse_stm2 stream
    | _ -> raise Not_found
and parse_stm1 stream = (* ID := E T *)
  let v = match lookahead stream with ID s -> s | _ -> raise Not_found in
  let e, stream_term = parse_exp (eat (eat stream (ID v)) ASSIGN) in
    Hashtbl.add variable v e;
    parse_t stream_term
and parse_stm2 stream = (* print ( F ) T *)
  let e, stream_term = parse_f (eat (eat stream PRINT) LPAR) in
    print_int e;
    parse_t (eat stream_term RPAR)
and parse_t stream =
  match lookahead stream with
    | SEMI -> parse_stm (eat stream SEMI)
    | COMMA | EOF -> stream  (* empty *)
    | _ -> raise Not_found
and parse_f stream =
  let e, stream_term = parse_exp stream in
    try parse_g stream_term  with Empty_sem -> e, stream_term
and parse_g stream =
  match lookahead stream with
    | COMMA -> parse_f (eat stream COMMA)
    | RPAR -> raise Empty_sem
    | _ -> raise Not_found
```

```
and parse_exp stream =
  match lookahead stream with
    | INT i -> parse_d (eat stream (INT i)) i
    | ID s  -> parse_ed (eat stream (ID s)) s
    | PRINT -> parse_exp (eat (parse_stm2 stream) COMMA)
    | LPAR  -> let e,s = parse_exp (eat stream LPAR) in parse_d (eat s RPAR) e
    | _ -> raise Not_found
and parse_ed stream varstr =
  match lookahead stream with
    | ASSIGN ->
        let e, stream_term = parse_exp (eat stream ASSIGN) in
          Hashtbl.add variable varstr e;
  parse_exp (eat (parse_t stream_term) COMMA)
    | _ -> parse_d stream (Hashtbl.find variable varstr)
and parse_d stream op1 =
  match lookahead stream with
    | PLUS  -> let e,s = parse_exp (eat stream PLUS) in (op1 + e, s)
    | MINUS -> let e,s = parse_exp (eat stream PLUS) in (op1 - e, s)
    | MUL   -> let e,s = parse_exp (eat stream PLUS) in (op1 * e, s)
    | DIV   -> let e,s = parse_exp (eat stream PLUS) in (op1 / e, s)
    | _ -> (op1, stream)
;;
let l = parse2stream " a := 6; a := (a := a+1, a+4) + a; print (a)"
;;
parse_prog l
```