Jacques Garrigue, 2006 年 11 月 1 日

II　言語処理系の構成法

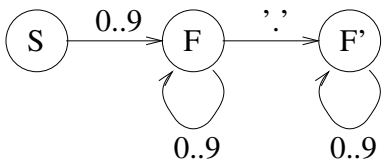
プログラムや構造化されたデータを文書として入力するとき，それを解析する必要がある．構造を扱いやすくするために，通常はそれを二段階で行なう．まず，入力を単語などを表す記号列に分ける．それを語句解析という．続いて，知られている文法によって，語の列に構造を入れる．それを構文解析という．

1　有限オートマトンと語句解析

有限オートマトンは最も簡単な計算仕組みの一つである．入力によって，いくつかの状態の間を移行する．表現力が限られているが，止まるかどうか (無限な長さの入力を受け取るか) 判断できるなど，よい性質を持っている．

基本的な形では，有限オートマトンは状態の集合，入力記号の集合，初期状態，入力記号と現在の状態から次の状態を決める関数表，終了状態の集合という 5 つ組で定義される．初期状態から始まり，毎回移行ができて，入力が終わったときに終了状態にいれば，その入力が受理されたという．途中で移行ができなかったり，または終わったときに終了状態以外の状態にいれば，受理されなかったという．

例えば，小数点数が次のような有限オートマトンで受理できる．



このオートマトンに対する関数表は以下のとおり．

	0..9	'.'	その他
S	F	E	E
F	F	F'	E
F'	F'	E	E

E は移行ができないときに行くエラー状態．

有限オートマトンは自然にリストに対する再帰関数で定義できる．

```
let rec s l =
  match l with
    ('0'..'9')::t -> f t                          (* 数字で f に移行 *)
  | _ -> false                                  (* それ以外は受理できない *)
and f l =
  match l with
    ('0'..'9')::t -> f t                          (* 数字で繰り返し *)
  | '.' :: t -> f' t                              (* '.' で f' に移行 *)
  | [] -> true                                    (* 終了状態なので *)
  | _  -> false                                 (* 上記以外に入力は受理できない *)
and f' l =
```

```
  match l with
    ('0'..'9')::t -> f' t
  | [] -> true                                          (* こちらも終了状態 *)
  | _  -> false
val s : char list -> bool
val f : char list -> bool
val f' : char list -> bool
```

文字列に対して上の関数を使いたい場合，以下のような変換関数が要る．

```
let explode s =
  let rec build n =
    if n < String.length s then s.[n] :: build (n+1) else []
  in build 0
val explode : string -> char list
```

そうすると，受理できるかどうかをチェックできる．

```
# let is_decimal str = s (explode str) ;;
val is_decimal : string -> bool = <fun>
# is_decimal "3.14";;
- : bool = true
# is_decimal ".5";;
- : bool = false
# is_decimal "33 ";;
- : bool = false
```

上記の再帰関数が有限オートマトンを表すために，処理の分岐にリストの先頭しか使わず，再帰呼び出しを一つだけ行うという条件が付く．

有限オートマトンで表現できる言語 (受理できる語の集合) は以下の演算に対して閉じている．

- 言語の連結
- 言語の任意回の繰り返し
- 言語の交わり
- 言語の補集合および言語の和

連結や繰り返しは状態を増やせずに行えるが，交わりは状態の積，補集合は状態の冪を必要とする．

理論的には受理できるかどうかということだけでいいが，実際のプログラムでは入力を解釈して出力を返したい．さらに，複数の語を続けて読みたい．上の関数に以下のような変更を加えるとそれができる．

```
let digit c = Char.code c - Char.code '0'
val digit : char -> int

let rec s l =
  match l with
    ('0'..'9' as c)::t -> f (digit c) t
  | _ -> failwith "not a number"                  (* 終了状態ではないので失敗 *)
and f n l =
  match l with
    ('0'..'9' as c)::t -> f (n * 10 + digit c) t   (* 読みながら値を計算する *)
  | '.' :: t -> f' n 1 t
  | _ -> (float n, l)                            (* 値と残りのリストの対を返す *)
```

```
and f' n m l =
  match l with
    ('0'..'9' as c)::t -> f' (n * 10 + digit c) (m * 10) t
  | _ -> (float n /. float m, l)
val s : char list -> float * char list
val f : int -> char list -> float * char list
val f' : int -> int -> char list -> float * char list

let rec decimal_list l =
  match l with
    [] -> []
  | ' '::t -> decimal_list t                          (* リストを空白で区切る *)
  | _ ->
      let (x, l') = s l in
      x :: decimal_list l'
val decimal_list : char list -> float list
```

これで小数点数のリストが解釈できる.

```
# decimal_list (explode "12.3 3 4.5");;
- : float list = [12.3; 3.; 4.5]
```

練習問題 1.1 *1.* `type number = Int of int | Float of float` という型定義を導入し，F で終了したときと F' で終了したときを区別せよ.

```
val decimal_list :  char list -> number list
```

2. `explode` の逆の働きをする関数 `implode` を定義しなさい．`String.create` と `String.set` を使えばいい.

```
val implode :  char list -> string
```

3. *OCaml* の識別子はアルファベット小文字または *'_'* で始まり，その後の文字ではアルファベット大文字・数字・*'' ''*が使える．識別子のリストの受理し，文字列のリストとして返す関数を定義しなさい.

```
val ident_list :  char list -> string list
```

2 ストリーム・パーザと構文解析

構文解析をやろうとすると，有限オートマトンの表現力では足りなくなる．有限オートマトンで表現できない簡単な例として，括弧が合っているかどうかの判定がある．(((())()) は合っているが，(((()))) は合っていない．具体的な入力をチェックするのに，その入力に現われる最大の入れ子の数の状態が必要だと証明できるので，任意の入力を扱おうとすると有限ではない.

有限オートマトンより強くて，かついい性質を持つ仕組みがいくつか知られている．その中でもプッシュダウン・オートマトンが任意の文脈自由文法を表現できる．しかし，一般的なプッシュダウン・オートマトンを実装すると効率が悪いので，通常はもっと制限されたものが使われる.

OCaml のストリーム・パーザはそんな仕組みの一つである．入力は有限オートマトンと同じような形で扱うが，リストではなく，ストリームというデータ構造を使う.

OCaml に附属している Camlp4 でストリーム・パーザと一緒に簡単な字句解析のライブラリーも提供されている.

```
# #load"camlp4o.cma";;
        Camlp4 Parsing version 3.09.2
```

```
# open Genlex;;
# let lexer = Genlex.make_lexer ["+";"*";"(";")"] ;;
val lexer : char Stream.t -> Genlex.token Stream.t = <fun>
# let s = lexer (Stream.of_string "1 2 3 4");;
val s : Genlex.token Stream.t = <abstr>
# (parser [< ' x >] -> x) s ;;                          (* 任意のトークンを読む *)
- : Genlex.token = Int 1
# (parser [< 'Int 1 >] -> "ok") s ;;                      (* Int 1だけを読む *)
Exception: Stream.Failure.                               (* 消されているので失敗 *)
# (parser [< 'Int 1 >] -> "one" | [< 'Int 2 >] -> "two") s ;;
- : string = "two"                                  (* 次のトークンはInt 2だった *)
```

練習問題 2.1 *1.* ストリームをリストに変える関数を定義せよ.

```
val list_of_stream :  'a stream -> 'a list
```

2. Genlex を使う代りに，前節の字句解析で定義した小数点数を *1* 個返す関数 *(*s, f, f'*)* の入力をストリームに変えなさい．*(*前問の list_of_stream を使わずに*)*

3. decimal_list を参考にして，number のストリームを返す関数を定義せよ.

```
decimal_stream :  char Stream.t -> number Stream.t
```

ストリームパーザと汎関数

ストリームパーザも汎関数と相性がいい.

```
(* リストをパーズしながら結果を蓄積 *)
# let rec accumulate parse accu = parser
    | [< e = parse accu; s >] -> accumulate parse e s
    | [< >] -> accu;;
val accumulate : ('a -> 'b Stream.t -> 'a) -> 'a -> 'b Stream.t -> 'a = <fun>
```

'e = parse accu' のようなパターンは，入力ストリーム s に対して関数適用 parse accu s を実行し，その結果を e に束縛する．もしも parse が Stream.Failure を起こしたら，次の場合に移る．しかし，Stream.Failure を起した関数がストリームパターンの先頭でなければならない.

```
(* 左結合の演算子を定義 *)
# let left_assoc parse op wrap =
    let parse' accu =
      parser [< 'Kwd k when k = op; s >] -> wrap accu (parse s) in
    parser [< e1 = parse; e2 = accumulate parse' e1 >] -> e2;;
val left_assoc :
  (Genlex.token Stream.t -> 'a) ->
  string -> ('a -> 'a -> 'a) -> Genlex.token Stream.t -> 'a = <fun>
```

'パターン when 条件' は条件が true になったときだけ選ばれる．通常のパターンマッチングでも使える.

汎関数を使うとパーザがとても短かく書ける.

```
# let rec parse_simple = parser
    | [< 'Int n >] -> Num n
    | [< 'Ident x >] -> Var x
    | [< 'Kwd"("; e = parse_expr; 'Kwd")" >] -> e
  and parse_mult s =
    left_assoc parse_simple "*" (fun e1 e2 -> Mult(e1,e2)) s
  and parse_expr s =
    left_assoc parse_mult "+" (fun e1 e2 -> Plus(e1,e2)) s ;;
```

```
val parse_simple : Genlex.token Stream.t -> expr = <fun>
val parse_mult : Genlex.token Stream.t -> expr = <fun>
val parse_expr : Genlex.token Stream.t -> expr = <fun>

# let parse_string s =
    match lexer (Stream.of_string s) with parser
      [< e = parse_expr; _ = Stream.empty >] -> e;;
val parse_string : string -> expr = <fun>
# let e = parse_string "5+x*(4+x)";;
val e : expr = Plus (Num 5, Mult (Var "x", Plus (Num 4, Var "x")))
# eval (subst ["x", 3] e);;
- : expr = Num 26
```

3 プリティ・プリンター

Format モジュールを使えば，式を綺麗に出力できる．

```
# let rec print_expr prio ppf e =
  let printf fmt = Format.fprintf ppf fmt in
  match e with
  | Num x -> printf "%d" x
  | Var x -> printf "%s" x
  | Mult (e1, e2) ->
      printf "@[%a *@ %a@]" (print_expr 1) e1 (print_expr 1) e2
  | Plus (e1, e2) as e ->
      if prio > 0 then printf "(%a)" (print_expr 0) e else
      printf "@[%a +@ %a@]" (print_expr 0) e1 (print_expr 0) e2;;
val print_expr : int -> Format.formatter -> expr -> unit
```

プリティ・プリンターをトップレベルで使うことができる．

```
# let print_expr0 = print_expr 0;;
val print_expr0 : Format.formatter -> expr -> unit = <fun>
# #install_printer print_expr0;;
# Plus(Num 3, Var "a");;
- : expr = 3 + a
```

4 read-eval-print ループ

簡単なインタープリタを作るには，パーザ・評価・プリンターを組み合わせればいい．

```
# let toplevel () =
  while true do
    try                                              (* エラー (例外) から守る *)
      print_string "? ";                                    (* ? を出力 *)
      let s = read_line() in                              (* 一行を読む *)
      let e = parse_string s in                          (* expr に変換 *)
      let e' = eval e in                                        (* 評価 *)
      Format.printf "=> %a@." (print_expr 0) e'                (* 出力 *)
    with                                          (* 例外のパターンマッチング *)
      Stream.Failure -> ()
    | Stream.Error _ -> Format.printf "Syntax error!@."
  done
val toplevel : unit -> unit = <fun>
```

```
# toplevel ();;
? 5+3*2
=> 11
? 5+3*a
=> 5 + 3 * a
? a+3*2
=> a + 6
? <C-c><C-c>Interrupted.
```

練習問題 4.1　*1.* 式の構文に –式 を追加し，パーザとプリンターも修正せよ．

2. 同様に ‘let 名前 = 式 in 式’という構文を追加せよ．そのために lexer を以下のように定義しなければならない．

```
let lexer = Genlex.make_lexer ["+";"*";"(";")";"=";"let";"in"] ;;
```

subst や eval の定義にも気を付けよ．トップレベルでの動作は以下のようになる．

```
# toplevel ();;
? let x = 1 + 1 in x + (let y = 1+2 in y*y)
=> 11
```