3 HTTP サーバの構築

今回は単純な機能をもった HTTP サーバを作る．

3.1 HTTP プロトコル

ウェブブラウザは HTTP プロトコルを使ってサーバと通信している．例えば，以下の URL をブラウザのアドレスバーに入れたとする．

```
http://localhost:6333/hello/world
```

まず，ブラウザが llocalhost のポート 6333 番に TCP でつなぎに行く．つながったら，以下のリクエストを送る．

```
GET /hello/world HTTP/1.1
Host: localhost:6333
User-Agent: Mozilla/5.0 (X11; U; FreeBSD i386; en-US; rv:1.8.1.1)
            Gecko/20061228 Firefox/2.0.0.1
Accept: text/xml,application/xml,application/xhtml+xml,
        text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: ja,en;q=0.8,en-us;q=0.5,fr;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

一行目はリクエストの最も重要な部分．最初にコマンド名が記述され，通常は GET になる．つづいて，要求された URL が入る．ホスト名やポート番号が含まれない．URL の中には空白が許されない．最後にプロトコル (HTTP) とバージョン番号が入る．

サーバの仕事は，このリクエストを解釈し，要求されたデータを送ることである．そのデータがウェブページであれば，HTML という言語で記述される．

3.2 HTML の構造

ブラウザで正しく表示されるために，HTML で返事をしなければならない．HTML の記述は比較的に簡単であるが，構造をもっていることに注意しなければならない．

```
<html>
  <!-- コメントはこう書く-->
  <head>
     <title>ウィンドウで表示されるタイトル</title>
  </head> <!-- 閉じるときに / を使う -->
  <body>
    <h1>ページの中の見出し</h1>
    <pre>
    こうすると，
    改行がそのままになる．
    </pre>
  </body>
</html>
```

3.3 最も単純なサーバ

このサーバはリクエストを HTML として返すだけだ．プログラムの構造は前に見た TCP サーバと基本的に同じである．ただ，プログラムを簡単にするために，read と write の代わりに input_line : in_channel -> string と output_string : out_channel -> string -> unit を使う．

```
(* http_server.ml *)
open Unix

(* 空行を見付けるまで入力を一行ずつ読み，読んだ分をリストとして返す *)
let rec read_lines ic =
  let l = input_line ic in                                    (* ic から一行を読む *)
  if l = "" || l = "\013" then []                              (* 空行なら空リスト *)
  else l :: read_lines ic                                       (* 違えば繰り返す *)
val read_lines : in_channel -> string list

let main () =
  let s = Unix.socket PF_INET SOCK_STREAM 0 in                 (* TCP サーバの準備 *)
  Unix.setsockopt s SO_REUSEADDR true;
  let server = ADDR_INET(inet_addr_any, 6333) in
  Unix.bind s server;
  Unix.listen s 5;

  while true do                                               (* 次のリクエストを待つ *)
    let rs, client = Unix.accept s in
    let ic = in_channel_of_descr rs                (* ソケットから ocaml の in_channel *)
    and oc = out_channel_of_descr rs in                    (* と out_channel を作る *)
    let req = input_line ic in                              (* 入力が行単位でできる *)
    let extra = read_lines ic in
    let ans =                                             (* 返事の HTML を構成する *)
      "<html><head><title>input</title></head>\n" ^
      "<body><h2>Request was:</h2>\n<pre>\n" ^
      String.concat "\n" (req :: extra) ^
      "\n</pre></body></html>\n"
    in
    Printf.printf "%s%!" ans;                          (* デバッグ用にローカルに出力 *)
    output_string oc ans;                                      (* ブラウザに送る *)
    flush oc;                                        (* 送られているのを確認する *)
    close rs;                                                  (* ソケットを閉じる *)
  done

let () =
  try main ()
  with Unix_error (err, f, s) ->
    Printf.eprintf "Unix error in %s: %s (%s)\n%!"
      f (Unix.error_message err) s
```

このプログラムをテストするために，まずサーバを走らせる．

```
$ ocaml unix.cma http_server.html
```

続いて，ウェブブラウザを開き，そのサーバを通る適当な URL を入力する．

```
http://localhost:6333/hello/world
```

結果は前ページのものと同様であるはずだ．

3.4 リクエストを解析する

HTTP は，リクエストの一行目だけでも様々なことができるように工夫されている．最も単純な場合では，URL がファイルを指し，それを送れば良い．

そのためにまず，ソケットに対してファイルを送る関数を定義する．

```
let send_file oc path =                              (* 引数は出力用ソケットとファイル名 *)
  try
    let ic = open_in path in                          (* ファイルを入力用として開く *)
    let n = ref 0 and buf = String.create 1024 in
    while n := input ic buf 0 1024; !n > 0 do              (* input は read と同様 *)
      output oc buf 0 !n               (* output は常に全データを送る（返り値は unit） *)
    done;
    close_in ic;
    ""                                                 (* 送れたときは空文字列を返す *)
  with Not_found ->
    "<h2>Unknown file = " ^ path ^ "</h2>\n"                  (* エラーメッセージ *)
val send_file : out_channel -> string -> string
```

この関数を利用するために main の一部を変える．

```
    let (cmd, url, ver) =                      (* req から文字列 2 つと float を取り出す *)
      Scanf.sscanf req "%s %s HTTP/%f" (fun c u v -> (c,u,v)) in
    Printf.printf "%s, %s, %g\n%!" cmd url ver;
    let res = if cmd = "GET" then send_file oc ("." ^ url) else "" in
    let ans =
      "<html><head><title>input</title></head>\n" ^
      "<body><h2>Request was:</h2>\n<pre>\n" ^
      String.concat "\n" (req :: extra) ^
      "\n</pre>\n" ^ res ^ "</body></html>\n"
    in
    Printf.printf "%s%!" ans;
    if res <> "" then output_string oc ans;                  (* エラーのときだけ *)
```

サーバを再起動し，3.2 節に従って hello.html というファイルを作る．そしてブラウザで以下の URL を見ると中身が表示されるはずだ．

```
http://localhost:6333/hello.html
```

3.5 HTML フォームの利用

ファイルを送るだけでは，動的な内容が作れない．サーバ内でページを生成するために，ブラウザから入力を受けなければならない．HTML のフォームがそれを可能にする．

calc.html というファイルに以下の中身を入れる．

```
<html>
<head><title>Calculator</title></head>
<body>
<h1>Calculator</h1>
<h2>Formula to compute</h2>
<form method=GET action="/calc">
  <textarea name=formula cols=40 rows=5></textarea><p>
  <input type=submit value="Send"> <input type=reset>
</form>
</body>
</html>
```

上と同様に，それをブラウザに読み込むと，入力フィールドが現われる．そこに (13+5)*4 を入力して，Send ボタンを押すと，URL は以下のようになる．

```
http://localhost:6333/calc?formula=%2813%2B5%29*4
```

/calc は上のフォームで指定した action である．?の後は引数で，それぞれに名前が付けられる．ここは引数は一個しかないが，一般的には+ で区切られる．そのために，引数の中では + がそのまま使えなくて，コード化しなければならない．ASCII 表記を 16 進数にすると 2B になり，上の %2B がそれである．

前のように字句解析をすることも考えられるが，とても簡単な場合なので直接に文字列を処理することにした．

```
let rec decode_arg ~pos s =
  if pos >= String.length s then [] else
  if s.[pos] = '%' then
    let c = Scanf.sscanf (String.sub s (pos+1) 2) "%x" Char.chr in
    c :: decode_arg ~pos:(pos+3) s
  else s.[pos] :: decode_arg ~pos:(pos+1) s
val decode_arg : pos:int -> string -> char list

let implode l =
  let s = String.create (List.length l) in
  let rec copy ~pos = function
      []   -> s
    | c::l -> s.[pos] <- c; copy ~pos:(pos+1) l
  in copy ~pos:0 l
val implode : char list -> string

let cgi_arg s =
  try
    let pos = String.index s '=' in                          (* 最初の = を見つける *)
    let name = String.lowercase (String.sub s 0 pos) in
    let arg = implode (decode_arg ~pos:(pos+1) s) in
    (name, arg)
  with Not_found -> (s, "")                                    (* = がなかった場合 *)
val cgi_arg : string -> string * string

let parse_url s =
  try
    let pos = String.index s '?' in
    let path = String.lowercase (String.sub s 0 pos) in
    let arg = cgi_arg (String.sub s (pos+1) (String.length s - pos - 1)) in
    (path, [arg])
  with Not_found -> (s, [])                                    (* ? がなかった場合 *)
val parse_url : string -> string * (string * string) list
```

最終的に，URL の中身によって動作を決める関数を定義する．

```
let process_url oc s =
  match parse_url s with
    "/calc", ["formula", e] ->                          (* フォームが送られて来たとき *)
      "<h2>Formula = " ^ e ^ "</h2>"
  | path, [] ->                                          (* ファイルが求められたとき *)
      send_file oc ("." ^ path)
  | cmd, _ ->                                                (* それ以外のコマンド *)
      "<h2>Unknown command = " ^ cmd ^ "</h2>"
val process_url : out_channel -> string -> string
```

main もまた変えなければならない．res を定義する行だけでいい．

```
    let res = if cmd = "GET" then process_url oc url else "" in
```

以上のプログラムで復号化された数式が表示されるようになる．

練習問題 3.1 *1.* 前に見た数式の構文解析を利用して，数式の結果を計算するようにプログラムを変更せよ．

2. 複数のフィールドを含むフォームを扱えるようにせよ．そのために + で区切られた文字列を文字列のリストに分ける関数を書かなければならない．ちなみに，一行だけのフィールドを定義したいとき，以下のように書けばいい．

```
<input name="フィールド名" value="初期値" size="文字数">
```