```
(*---------------------------------------------------------------------
 *
 * James Vrionis
 * Luke Tanner
 * CMPS112
 * ASG2
 *
 *---------------------------------------------------------------------*)

module Bigint : sig
   type bigint
   val bigint_of_string : string -> bigint
   val string_of_bigint : bigint -> string
   val add : bigint -> bigint -> bigint
   val sub : bigint -> bigint -> bigint
   val mul : bigint -> bigint -> bigint
   val div : bigint -> bigint -> bigint
   val rem : bigint -> bigint -> bigint
   val pow : bigint -> bigint -> bigint
   val zero : bigint
end
```

```
(*------------------------------------------------------------------
 *
 * James Vrionis
 * Luke Tanner
 * CMPS112
 * ASG2
 *
 *------------------------------------------------------------------*)

open Printf

module Bigint = struct

    type sign     = Pos | Neg
    type bigint   = Bigint of sign * int list
    let  radix    = 10
    let  radixlen =  1
(*------------------------------------------------------------------*)

    let car   = List.hd
    let cdr   = List.tl
    let map   = List.map
    let reverse   = List.rev
    let strcat    = String.concat
    let strlen    = String.length
    let strsub    = String.sub
    let zero      = Bigint (Pos, [])

(*--------------------------------------------------------------------
  Char list of Strings:
--------------------------------------------------------------------*)
    let charlist_of_string str =
        let last = strlen str - 1
        in  let rec charlist pos result =
            if pos < 0
            then result
            else charlist (pos - 1) (str.[pos] :: result)
        in  charlist last []
    ;;


(*--------------------------------------------------------------------
   BigInt of strings (in-order):
--------------------------------------------------------------------*)
    let bigint_of_string str =
        let len = strlen str
        in  let to_intlist first =
                let substr = strsub str first (len - first) in
                let digit char = int_of_char char - int_of_char '0' in
                map digit (reverse (charlist_of_string substr))
            in  if   len = 0 then zero
                else if   str.[0] = '_'
                     then Bigint (Neg, to_intlist 1)
                     else Bigint (Pos, to_intlist 0)
    ;;

(*--------------------------------------------------------------------
   BigInt of strings (reversed):
```

```
--------------------------------------------------------------------*)
    let string_of_bigint (Bigint (sign, value)) =
        match value with
        | []    -> "0"
        | value -> let reversed = reverse value
                   in  strcat ""
                       ((if sign = Pos then "" else "-") ::
                        (map string_of_int reversed))
    ;;


(*--------------------------------------------------------------------
  Recursive Subtraction Function:
   (CTF) stands for call to function.
--------------------------------------------------------------------*)
   let rec sub' list1 list2 steal = match (list1, list2, steal) with
     | list1, [], 0 -> list1
     | [], list2, 0 -> failwith "Valid only if list2 > list"
     | car1::cdr1, [], steal  ->
         if car1 = 0 then 9 :: (sub' cdr1 [] 1)
         else let dif = car1 - steal*1 in dif :: (sub' cdr1 [] 0)
     | [], list2, steal -> failwith "Err in sub':Invalid CTF"
     | car1::cdr1, car2::cdr2, steal ->
         if car2 > (car1 - steal*1) then
         let dif = ((car1 + 10) - steal*1) - car2
                 in dif :: (sub' cdr1 cdr2 1)
         else let dif = (car1 - steal*1) - car2
                 in dif :: (sub' cdr1 cdr2 0)

(*--------------------------------------------------------------------
    Recursive Add():
--------------------------------------------------------------------*)
    let rec add' list1 list2 carry = match (list1, list2, carry) with
        | list1, [], 0       -> list1
        | [], list2, 0       -> list2
        | list1, [], carry   -> add' list1 [carry] 0
        | [], list2, carry   -> add' [carry] list2 0
        | car1::cdr1, car2::cdr2, carry ->
          let sum = car1 + car2 + carry
          in  sum mod radix :: add' cdr1 cdr2 (sum / radix)


(*--------------------------------------------------------------------
    Recursive sub():
--------------------------------------------------------------------*)
    let rec sub' list1 list2 borrow = match(list1, list2, borrow) with
        | [], _, _           -> []
        | list1, [], 0       -> list1
        | list1, [], borrow   -> sub' list1 [borrow] 0
        | car1::cdr1, car2::cdr2, borrow ->
          let dif = car1 - borrow - car2
          in (if dif < 0 then dif + 10 :: sub' cdr1 cdr2 1
            else dif :: sub' cdr1 cdr2 0)

(*--------------------------------------------------------------------
  Compare Recursively Defined:
   Bool -> Bool ->  Bool
   && is LR: e1 && e2, e1 is evaluated first (if false) e2 is not
```

```
   evaluated
---------------------------------------------------------------------*)
    let rec cmp' list1 list2 = match (list1, list2) with
        | [], []                 ->  0
        | list1, []              ->  1
        | [], list2              -> -1
        | car1::cdr1, car2::cdr2 ->
            let result = cmp' cdr1 cdr2
            in if result = 0 && car1 <> car2 then
               if car1 > car2 then 1
               else if car1 < car2 then -1
               else 0
            else result
    ;;

(*---------------------------------------------------------------------
  Into Base Value:
---------------------------------------------------------------------*)
    let rec into base value count =
        if (cmp' value base = 1) then base, 0
        else let check = add' value value 0
            in if (cmp' check base = 1) then value, count
        else into base check (count+1)
    ;;

(*---------------------------------------------------------------------
  Time MSBs():
---------------------------------------------------------------------*)
    let trimzeros list =
        let rec trimzeros' list' = match list' with
            | []       -> []
            | [0]      -> []
            | car::cdr ->
                let cdr' = trimzeros' cdr
                in  match car, cdr' with
                    | 0, [] -> []
                    | car, cdr' -> car::cdr'
            in trimzeros' list
    ;;

(*---------------------------------------------------------------------
 Recursive value doubling:

---------------------------------------------------------------------*)
    let rec doubler value count = match (value, count) with
        | value, 0      -> value
        | value, count  -> (doubler (add' value value 0) (count-1))
    ;;

(*---------------------------------------------------------------------
  Recursive Division():

---------------------------------------------------------------------*)
    let rec divrem' dividend divisor sum =
        if (cmp' dividend [] = 0) then sum, [0]
        else let num, count = into dividend divisor 1
        in if count = 0 then sum, dividend
        else divrem' (trimzeros (sub' dividend num 0))
```

```
                    divisor (add' sum (doubler [1] (count - 1)) 0)
    ;;


(*----------------------------------------------------------------------
    Recursive Multiply():

-----------------------------------------------------------------------*)
   let rec mul' value base sum = match (value, base, sum) with
        | [], _, sum          -> sum
        | [1], base, sum  -> add' base sum 0
        | value, base, sum ->
            let num, count = into value [2] 1
            in mul' (trimzeros (sub' value num 0)) base
            (add' sum (doubler base count) 0)

(*----------------------------------------------------------------------
  Recursive Exponentiation():

-----------------------------------------------------------------------*)
let rec expt value count = match (value, count) with
        | value, 0      -> value
        | value, count  -> (expt (mul' value value []) (count-1))
    ;;

(*----------------------------------------------------------------------
  Recursive Power():

-----------------------------------------------------------------------*)
    let rec pow' expo base prod = match (expo, base, prod) with
        | [], _, prod      -> Pos, prod
        | [1], base, prod  -> Neg, mul' base prod []
        | expo, base, prod ->
            let num, count = into expo [2] 1
            in pow' (trimzeros (sub' expo num 0))
            base (mul' prod (expt base count) [])
    ;;

(*----------------------------------------------------------------------
  Compare:
    if the sign is the same then Compare
-----------------------------------------------------------------------*)
    let cmp (Bigint (neg1, arg1)) (Bigint (neg2, arg2)) =
        if neg1 = neg2
            then cmp' arg1 arg2
        else if neg1 = Pos
            then 1
            else -1
    ;;

(*----------------------------------------------------------------------
  Add():
      The process or skill of calculating the total of two or
      more numbers or amounts.
-----------------------------------------------------------------------*)
    let add (Bigint(neg1, arg1)) (Bigint(neg2, arg2)) =
        if neg1 = neg2 then Bigint(neg1, add' arg1 arg2 0)
        else if(cmp' arg1 arg2) = 1
```

```
            then Bigint(neg1, (trimzeros(sub' arg1 arg2 0) ))
        else if(cmp' arg1 arg2) = -1
            then Bigint(neg2, (trimzeros(sub' arg2 arg1 0) ))
        else zero
    ;;

(*---------------------------------------------------------------------
  Subtraction():
      Anal -> analyze
---------------------------------------------------------------------*)
 let sub (Bigint (neg1, arg1)) (Bigint (neg2, arg2)) =
        if neg1 = neg2 then
            let anal = cmp' arg1 arg2 in
                 if anal > 0
                    then Bigint (neg1, trimzeros (sub' arg1 arg2 0))
                 else if anal < 0 then
                     let sign = if neg1 = Pos then Neg else Pos
                     in  Bigint (sign, trimzeros (sub' arg2 arg1 0))
                 else zero
        else Bigint (neg1, add' arg1 arg2 0)


(*---------------------------------------------------------------------
  Multiply():
      Anal -> analyze
      combine quantities under given rules to obtain their product.
---------------------------------------------------------------------*)
    let mul (Bigint (neg1, arg1)) (Bigint (neg2, arg2)) =
        let anal = (cmp' arg1 arg2 = 1) in (if neg1 = neg2
            then (if anal
                then Bigint (Pos, mul' arg2 arg1 [])
                else Bigint (Pos, mul' arg1 arg2 []))
            else (if anal
                then Bigint (Neg, mul' arg2 arg1 [])
                else Bigint (Neg, mul' arg1 arg2 [])))
    ;;

(*---------------------------------------------------------------------
  Power():
      The number of times as indicated by an exponent that a number
      occurs as a factor in a product
---------------------------------------------------------------------*)
    let pow (Bigint (neg1, arg1)) (Bigint (neg2, arg2)) =
        let sign, value = pow' arg2 arg1 [1]
        in if neg1 = Pos
            then Bigint (Pos, value)
            else Bigint (sign, value)
    ;;

(*---------------------------------------------------------------------
  Division/Remainder():
      Return the remainder of 2 terms by division

---------------------------------------------------------------------*)
    let divrem (Bigint (neg1, arg1)) (Bigint (neg2, arg2)) =
    let quot, xcess = divrem' arg1 arg2 []
        in match (neg1, neg2) with
            | Pos, Pos -> Bigint (Pos, quot), Bigint (Pos, xcess)
```

```
            | Neg, Pos -> Bigint (Neg, add' quot [1] 0),
                          Bigint (Pos, trimzeros (sub' arg2 xcess 0))
            | Pos, Neg -> Bigint (Neg, quot), Bigint (Pos, xcess)
            | Neg, Neg -> Bigint (Pos, add' quot [1] 0),
                          Bigint (Pos, trimzeros (sub' arg2 xcess 0))
    ;;

(*--------------------------------------------------------------------
  Remainder():
      Return the remainder by division
--------------------------------------------------------------------*)
    let rem (Bigint (neg1, arg1)) (Bigint (neg2, arg2)) =
        let _, remainder = divrem (Bigint (neg1, arg1))
                                  (Bigint (neg2, arg2))
        in remainder
    ;;

(*--------------------------------------------------------------------
  Division():
      Return the Quotient of a Bigint by Division
--------------------------------------------------------------------*)
    let div (Bigint (neg1, arg1)) (Bigint (neg2, arg2)) =
        let quotient, _ = divrem (Bigint (neg1, arg1))
                                 (Bigint (neg2, arg2))
        in quotient
    ;;
(*------------------------------------------------------------------*)
end
```

```
(*------------------------------------------------------------------
 *
 * James Vrionis
 * Luke Tanner
 * CMPS112
 * ASG2
 *
 *------------------------------------------------------------------*)

include Scanner
include Bigint

open Bigint
open Printf
open Scanner

type stack_t = Bigint.bigint Stack.t
let push = Stack.push
let pop = Stack.pop

let ord thechar = int_of_char thechar
type binop_t = bigint -> bigint -> bigint

let print_number number = printf "%s\n%!" (string_of_bigint number)

let print_stackempty () = printf "stack empty\n%!"

let executereg (thestack: stack_t) (oper: char) (reg: int) =
    try match oper with
        | 'l' -> printf "operator l reg 0%o is unimplemented\n%!" reg
        | 's' -> printf "operator s reg 0%o is unimplemented\n%!" reg
        | _   -> printf "0%o 0%o is unimplemented\n%!" (ord oper) reg
    with Stack.Empty -> print_stackempty()

let executebinop (thestack: stack_t) (oper: binop_t) =
    try let right = pop thestack
        in  try let left = pop thestack
                in  push (oper left right) thestack
            with Stack.Empty -> (print_stackempty ();
                                 push right thestack)
    with Stack.Empty -> print_stackempty ()

let execute (thestack: stack_t) (oper: char) =
    try match oper with
        | '+'  -> executebinop thestack Bigint.add
        | '-'  -> executebinop thestack Bigint.sub
        | '*'  -> executebinop thestack Bigint.mul
        | '/'  -> executebinop thestack Bigint.div
        | '%'  -> executebinop thestack Bigint.rem
        | '^'  -> executebinop thestack Bigint.pow
        | 'c'  -> Stack.clear thestack
        | 'd'  -> push (Stack.top thestack) thestack
        | 'f'  -> Stack.iter print_number thestack
        | 'l'  -> failwith "operator l scanned with no register"
        | 'p'  -> print_number (Stack.top thestack)
        | 'q'  -> raise End_of_file
        | 's'  -> failwith "operator s scanned with no register"
        | '\n' -> ()
```

```
        | ' '  -> ()
        | _    -> printf "0%o is unimplemented\n%!" (ord oper)
    with Stack.Empty -> print_stackempty()

let toploop (thestack: stack_t) inputchannel =
    let scanbuf = Lexing.from_channel inputchannel in
    let rec toploop () =
        try  let nexttoken = Scanner.scanner scanbuf
             in  (match nexttoken with
                 | Number number       -> push number thestack
                 | Regoper (oper, reg) -> executereg thestack oper reg
                 | Operator oper       -> execute thestack oper
                 );
             toploop ()
        with End_of_file -> printf "End_of_file\n%!";
    in  toploop ()

let readfiles () =
    let thestack : bigint Stack.t = Stack.create ()
    in  ((if Array.length Sys.argv > 1
         then try  let thefile = open_in Sys.argv.(1)
                   in  toploop thestack thefile
              with Sys_error message -> (
                   printf "%s: %s\n%!" Sys.argv.(0) message;
                   exit 1));
        toploop thestack stdin)

let interact () =
    let thestack : bigint Stack.t = Stack.create ()
    in  toploop thestack stdin

let _ = if not !Sys.interactive then readfiles ()

```

```
(*----------------------------------------------------------------------
 *
 * James Vrionis
 * Luke Tanner
 * CMPS112
 * ASG2
 *
 *----------------------------------------------------------------------*)

{

module Scanner = struct
    include Bigint

    type token = Number   of Bigint.bigint
               | Regoper  of char * int
               | Operator of char

    let bigstr = Bigint.bigint_of_string
    let lexeme = Lexing.lexeme
    let ord    = int_of_char
    let strlen = String.length

    let regoper lexbuf =
        let token = lexeme lexbuf
        in  Regoper (token.[0], ord token.[1])

}

let number  = '_'? ['0' - '9']*
let regoper = ['s' 'l']

rule scanner = parse
   | number    { Number (bigstr (lexeme lexbuf)) }
   | regoper _ { regoper lexbuf }
   | _         { Operator (lexeme lexbuf).[0] }
   | eof       { raise End_of_file }

{

end

}
```

```
(* $Id: dc.ml,v 1.1 2011-04-26 13:39:18-07 - - $ *)

(*
* This file is useless for compilation.  However, for interactive
* testing it make loading all three files easier.  Normally for
* interactive use, type
*
*    #use "dc.ml";;
*
* at the toplevel.  Alternately, to run it directly without
* interacting with the toplevel, just use:
*
*    ocaml dc.ml
*
* which will run the program without need for compilation.
*)

#use "bigint.ml";;
#use "scanner.ml";;
#use "maindc.ml";;
```

```
#   James Vrionis
#   Luke Tanner
#   CMPS112
#   ASG2



MKFILE    = Makefile
DEPSFILE  = ${MKFILE}.deps
NOINCLUDE = ci clean spotless
NEEDINCL  = ${filter ${NOINCLUDE}, ${MAKECMDGOALS}}
SUBMAKE   = ${MAKE} --no-print-directory

SOURCE    = bigint.mli bigint.ml maindc.ml scanner.mll
ALLSRC    = ${SOURCE} dc.ml ${MKFILE}
OBJCMO    = bigint.cmo scanner.cmo maindc.cmo
OBJCMI    = ${patsubst %.cmo, %.cmi, ${OBJCMO}}
CAMLRUN   = ocamldc
LISTING   = Listing.ps

all : ${CAMLRUN}

${CAMLRUN} : ${OBJCMO} ${OBJCMI}
        ocamlc ${OBJCMO} -o ${CAMLRUN}

%.cmi : %.mli
        ocamlc -c $<

%.cmo : %.ml
        ocamlc -c $<

%.ml : %.mll
        ocamllex $<

clean :
        - rm ${OBJCMO} ${OBJCMI} ${DEPSFILE} scanner.ml ocamldc
        - rm *.log *.ocamldcout *.dcout

spotless : clean
        - rm ${CAMLRUN} ${LISTING} ${LISTING:.ps=.pdf}

ci : ${RCSFILES}
        cid + ${ALLSRC}
        checksource ${ALLSRC}

deps : ${SOURCE}
        ocamldep ${SOURCE} >${DEPSFILE}

${DEPSFILE} :
        @ touch ${DEPSFILE}
        ${SUBMAKE} deps

lis : ${ALLSRC}
        mkpspdf ${LISTING} ${ALLSRC} ${DEPSFILE}

again :
        ${SUBMAKE} spotless ci deps
        ${SUBMAKE} all lis
```

```

ifeq (${NEEDINCL}, )
include ${DEPSFILE}
endif

.PRECIOUS : scanner.ml
```

```
bigint.cmi :
bigint.cmo : bigint.cmi
bigint.cmx : bigint.cmi
maindc.cmo : bigint.cmi
maindc.cmx : bigint.cmx
```