CIS 500 Software Foundations (Fall 2002)

Homework Assignment 2

More OCaml Programming

Due: Thursday, September 19, 2002, by 4PM

1 Exercise [Optional] Define a new type `complex` representing complex numbers. Write the following functions that operate on your new type:

- `real_part` : extracts the real part of the complex number
- `imag_part` : extracts the imaginary part of the complex number
- `add` : adds two complex numbers
- `mult` : multiplies two complex numbers
- `norm` : returns the norm of a complex number (the square root of the sum of the squares of the components - use the standard OCaml function `sqrt`)

(Don't worry if you don't remember the formulas for these things—this course is not about complex analysis! Just ask somebody.)
Examples:

```
# let c1 = Comp(2.0,3.0);;
val c1 : complex = Comp (2, 3)

# let c2 = Comp(5.0,1.0);;
val c2 : complex = Comp (5, 1)

# real_part c1;;
- : float = 2

# imag_part c1;;
- : float = 3

# norm c1;;
- : float = 3.60555127546

# add c1 c2;;
- : complex = Comp (7, 4)

# mult c1 c2;;
- : complex = Comp (7, 17)
```

2 Exercise Consider the following datatype of *tokens*:

```
type token =
    Num of int
  | Plus
  | Minus
  | Times
  | LParen
  | RParen
```

Write a function `lex` that takes a list of characters as input and produces a list of `token`s as output. Your function should:

- map sequences of digits to appropriate instances of the `Num` constructor
- map the characters `'+'`, `'-'`, `'*'`, `'('`, and `')'` to `Plus`, `Minus`, `Times`, `LParen`, and `RParen`, respectively
- ignore whitespace (the `' '` and `'\n'` characters)
- fail (by raising the exception `Bad`) on all other characters

Examples:

```
# lex ['(';'1';'2';'+';'3';'4';'0';')';' '];;
- : token list = [LParen; Num 12; Plus; Num 340; RParen]

# lex ['+';' ';'*'];;
- : token list = [Plus; Times]

# lex ['a'];;
Exception: Bad.

# lex [];;
- : token list = []

# lex ['(';'(';'1';'2';'+';'3';'4';'0';')';'*';' ';' ';'\n';'5';')'];;
- : token list =
      [LParen; LParen; Num 12; Plus; Num 340; RParen; Times; Num 5; RParen]
```

3 Exercise Here is a very simple grammar of fully parenthesized arithmetic expressions,

exp	::=	n	number
		$(\text{exp} + \text{exp})$	parenthesized sum of expressions
		$(\text{exp} - \text{exp})$	parenthesized difference of expressions
		$(\text{exp} * \text{exp})$	parenthesized product of expressions

and here is a datatype definition representing the corresponding type of abstract syntax trees (which we saw in class on Wednesday).

```
type ast =
    ANum of int
  | APlus of ast * ast
  | AMinus of ast * ast
  | ATimes of ast * ast;;
```

Write a function `parse` that takes a list `l` of tokens and produces a pair `(e,l')`, where `e` is a value of type `ast` (following the above grammar) and `l'` is a list of tokens representing the portion of `l` that was left over after parsing `e`. Your function should raise the exception `Bad` if the token list does not correspond to a legal expression.
Examples:

```
# parse [Num 50];;
- : ast * token list = (ANum 50, [])

# parse [LParen; Num 50];;
Exception: Bad.

# parse [LParen; Num 12; Plus; Num 340; RParen];;
- : ast * token list = (APlus (ANum 12, ANum 340), [])

# parse [LParen; LParen; Num 12; Plus; Num 340; RParen; Times; Num 5; RParen];;
- : ast * token list = (ATimes (APlus (ANum 12, ANum 340), ANum 5), [])

# parse [LParen; Num 12; Plus; Num 340; RParen; Times; Num 5];;
- : ast * token list = (APlus (ANum 12, ANum 340), [Times; Num 5])
```

4 Exercise [**Optional**] Put all of the pieces together: take the `eval` function given in lecture together with your `lex` and `parse` functions and write a function `calc` that takes a string and returns an integer. If the string represents a valid arithmetic expression, `calc` function should return its value as computed by `eval`. If it is not a valid expression, it should raise the exception `Bad`.
Examples:

```
# calc "((1+2)*3)";;
- : int = 9

# calc "(1+2) 5";;
Exception: Bad.

# calc "((2+1) * (11+8))";;
- : int = 57
```

You'll probably need the function `charl_from_string`, defined below:

```
let rec charl_from_string s =
  match s with
    "" -> []
  | _ -> (String.get s 0)::
           (charl_from_string (String.sub s 1 ((String.length s)-1)))
```

5 Debriefing

1. How many hours did you spend on this assignment?
2. Would you rate it as easy, moderate, or difficult?
3. Did you work on it mostly alone, or mostly with other people?
4. How deeply do you feel you understand the material it covers (0%–100%)?
5. Any other comments?

Solutions

1.

```
(* Note that we've chosen to use rectangular coordinates rather than
   polar; rectangular is simpler for the operations we want here. *)
type complex = Comp of float * float

let real_part = function
  Comp(x,_) -> x

let imag_part = function
  Comp(_,y) -> y

let add c1 c2 =
  match c1,c2 with
    Comp(x1,y1),Comp(x2,y2) -> Comp((x1+.x2),(y1+.y2))

let mult c1 c2 =
  match c1,c2 with
    Comp(x1,y1),Comp(x2,y2) -> Comp(((x1*.x2)-.(y1*.y2)),((x1*.y2)+.(y1*.x2)))

let norm = function
  Comp(x,y) -> sqrt((x*.x)+.(y*.y))
```

2.

```
type token =
    Num of int
  | Plus
  | Minus
  | Times
  | LParen
  | RParen

exception Bad

let rec lex s =
  match s with
    [] -> []
  | x::rest ->
      match x with
        ' ' | '\n' -> lex rest
      | '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' -> lexn s
      | '+' -> Plus :: (lex rest)
      | '-' -> Minus :: (lex rest)
      | '*' -> Times :: (lex rest)
      | '(' -> LParen :: (lex rest)
      | ')' -> RParen :: (lex rest)
      | _ -> raise Bad

and lexn s =
  let rec loop acc s' =
    match s' with
```

```
      [] ->
        [Num acc]
    | x::rest ->
        let digit d = loop (acc*10 + d) rest in
        match x with
          '0' -> digit 0
        | '1' -> digit 1
        | '2' -> digit 2
        | '3' -> digit 3
        | '4' -> digit 4
        | '5' -> digit 5
        | '6' -> digit 6
        | '7' -> digit 7
        | '8' -> digit 8
        | '9' -> digit 9
        | _ -> (Num acc) :: lex s'
  in loop 0 s;;
```

3.

```
type ast =
    ANum of int
  | APlus of ast * ast
  | AMinus of ast * ast
  | ATimes of ast * ast;;

let rec parse l =
  match l with
    (Num i) :: rest -> (ANum i, rest)
  | LParen::rest ->
      (let (e1,rest1) = parse rest in
       let (op,restop) = match rest1 with o::r -> (o,r) | [] -> raise Bad in
       let (e2,rest2) = parse restop in
       let e =
         match op with
           Plus -> APlus(e1,e2)
         | Minus -> AMinus(e1,e2)
         | Times -> ATimes(e1,e2)
         | _ -> raise Bad in
       match rest2 with
         RParen::rest3 -> (e, rest3)
       | _ -> raise Bad)
  | _ -> raise Bad;;
```

4.

```
let calc s =
  let parsed_result = parse (lex (charl_from_string s))
  in
    match parsed_result with
      (tree,[]) -> eval tree
    | _ -> raise Bad
```