CIS 500 — Software Foundations

Homework Assignment 2

More OCaml

Due: Monday, September 18, 2006, by noon

Instructions: Use the same submission procedure as last time, paying attention to the following points:

- Put all of your solutions together in a single OCaml source file named `hw2.ml`.
- Submit this file as `hw2`, for example, using the command:

```
~cis500/bin/cis500submit hw2 hw2.ml
```

- Anything that isn't valid OCaml code should be placed in a comment. We want to be able to run your file directly.

Reading assignment: Before beginning the programming exercises below, read Chapter 6 of Jason Hickey's *Introduction to Objective Caml*.

1 Exercise Consider the following datatype of *tokens*:

```
type token =
    Num of int
  | Plus
  | Minus
  | Times
  | LParen
  | RParen
  | If
  | Then
  | Else
  | And
  | Or
  | Equal
```

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`, `Equal`, `LParen`, and `RParen`, respectively
- map the two-character sequence `'&';'&'` to the token `And`, the sequence `'|';'|'` to the token `Or`, the sequence `'i';'f'` to the token `If`, the sequence `'t';'h';'e';'n'` to the token `Then`, and the sequence `'e';'l';'s';'e'` to the token `Else`.
- 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 ['if';' ';'5';'t';'h';'e';'n'];;
- : token list = [If; Num 5; Then]

# 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]
```

2 Exercise Here is a very simple grammar of fully parenthesized arithmetic and boolean expressions (extending the one we saw in class),

exp	::=	n	number
		(exp + exp)	parenthesized sum of expressions
		(exp − exp)	parenthesized difference of expressions
		(exp ∗ exp)	parenthesized product of expressions
		(exp = exp)	parenthesized comparison of expressions
		(exp`&&`exp)	parenthesized conjunction of expressions
		(exp`||`exp)	parenthesized disjunction of expressions
		`if` exp `then` exp `else` exp	conditional

and here is a datatype definition representing the corresponding set of abstract syntax trees.

```
type ast =
    ANum of int
```

```
    | APlus of ast * ast
    | AMinus of ast * ast
    | ATimes of ast * ast
    | AAnd of ast * ast
    | AOr of ast * ast
    | AEqual of ast * ast
    | AIf of ast * ast * ast
```

Evaluation of such expressions can yield either a number or a boolean; here is a datatype that captures both of these possibilities:

```
type value =
    Int of int
  | Bool of bool
```

Extend the function `eval` presented in class so that it deals with the extra constructs introduced in the grammar above. Your `eval` function should have type `ast -> value`.

For example:

```
# eval (AEqual ((ATimes (APlus (ANum 12, ANum 340), ANum 5)), (ANum 1760)));;
- : value = Bool true

# eval (AIf (AEqual (ANum 4, ANum 3),
             ANum 5,
             APlus (ANum 2, ANum 2)));;
- : value = Int 4
```

3 Exercise Here is a function `parse` that takes lists of `token`s and yield the corresponding `ast`:

```
let rec parse l =
  let parseToken t l =
    match l with
      [] -> raise Bad
    | x::rest -> if x=t then rest else raise Bad in
  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)
         | And -> AAnd(e1,e2)
         | Or -> AOr(e1,e2)
         | Equal -> AEqual(e1,e2)
```

```
          | _ -> raise Bad in
        match rest2 with
          RParen::rest3 -> (e, rest3)
        | _ -> raise Bad)
  | If::rest ->
       (let (e1,rest1) = parse rest in
        let rest2 = parseToken Then rest1 in
        let (e2,rest3) = parse rest2 in
        let rest4 = parseToken Else rest3 in
        let (e3,rest5) = parse rest4 in
        (AIf(e1,e2,e3), rest5))
  | _ -> raise Bad
```

And here is a function `explode` that takes a string and returns a list of the characters it contains:

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

Put all of these pieces together: take the `eval` function from the previous exercise, the `parse` and `explode` functions above, and your `lex` function from the first exercise, and write a function `calc` that takes a string and returns an integer. If the string represents a valid arithmetic expression, the `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 = Int 9

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

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

# calc "(3=(1+2))";;
- : value = Bool true

# calc "if (3=4) then (3+6) else (5*200)";;
- : value = Int 1000
```

4 **Exercise [Required for all groups (of any size, including 1) containing at least one PhD student; optional otherwise]** Extend your parser to handle unparenthesized expressions, using the usual rules of precedence (`&&` and `||` have lower precedence than `=`, which has lower precedence than `+` and `-`, which have lower precedence than `*`) and associativity (`1+2+3` means `(1+2)+3`).

5 Exercise

The `exists` function takes a predicate `p` (a one-argument function returning a boolean) and a list `l` and checks whether there is some element of `l` for which `p` returns true.

```
# exists (fun x -> x >= 3) [2;11;4];;
- : bool = true

# exists (fun x -> x >= 3) [1;1;2];;
- : bool = false
```

Define `exists` as a recursive function.

6 Exercise Give an alternate definition of `exists` using `fold` and *without* any other use of recursion.

7 Exercise Define `map` using `fold` (and without any other use of recursion).

8 Exercise Instead of defining the `stream` data type as we did in the lecture,

```
type 'a stream = Stream of 'a * (unit -> 'a stream);;
```

suppose we defined it like this:

```
type 'a stream = Stream of (unit -> ('a * 'a stream));;
```

Under this new definition, a stream does not "pre-compute" its first element. Instead, it waits until asked to compute anything at all; only then does it compute and return a head element and a new stream.

Rewrite *all* of the definitions of stream processing functions and examples (including the sieve of prime numbers) from the lecture notes so that they work with this new version of streams.

9 Debriefing

1. Approximately how many hours *per person* (on average) did you spend on this assignment?
2. Would you rate it as easy, moderate, or difficult?
3. How deeply do you feel you understand the material it covers (0%–100%)?
4. Any other comments?