CS 496: Extra Credit Assignment
Due: 26 March 2023, 11:55pm

1 Assignment Policies

Collaboration Policy. This assignment can be done in groups of at most two students. It is acceptable for students to collaborate in understanding the material but not in solving the problems or programming. Use of the Internet is allowed, but should not include searching for existing solutions.

Under absolutely no circumstances code can be exchanged between students. Excerpts of code presented in class can be used.

Assignments from previous offerings of the course must not be re-used. Violations will be penalized appropriately.

Be sure to update the parser from the PLaF repository, build and install it

2 Assignment

This assignment consists in extending REC to support mutually recursive function definitions. The resulting language will be called RECM in these notes.

2.1 Concrete Syntax of RECM

The concrete syntax of RECM results from modifying just one production in the concrete syntax of REC. The production in REC:

<Expression> ::= `letrec` <Identifier>(<Identifier>)=<Expression> `in` <Expression>

is replaced with:

<Expression> ::= `letrec` {<Identifier>(<Identifier>)=<Expression>}$^+$ `in` <Expression>

in RECM. The expression {<Identifier>(<Identifier>)=<Expression>}$^+$ above means that there may be 1 or more declarations. Here is an example of a valid program in RECM. Note how it declares two mutually recursive functions, namely `even` and `odd`:

```
letrec
  even(x) = if zero?(x)
              then zero?(0)
              else (odd (x - 1))
  odd(x)  = if zero?(x)
              then zero?(1)
              else (even (x - 1))
in (odd 99)
```

Evaluating this expression should produce the result `BoolVal true`, meaning that 99 is indeed odd. If we replace 99 in the code above with 98 and evaluate the resulting expression, this time we should get `BoolVal false` as a result. This is correct since 98 is not an odd number.

Fibonacci does not require mutual exclusion, but we can modify it slightly to produce another example of a program in RECM:

```
letrec
  fib2(n) = (fib (n-2))
  fib1(n) = (fib (n-1))
  fib(n) =
    if zero?(n)
    then 0
    else (if zero?(n-1)
          then 1
          else (fib1 n) + (fib2 n))
in (fib 10)
```

Evaluating this expression will produce `NumVal 55`, as expected.

3 Updated Parser for RECM

The parser has been updated so that it is capable of parsing expressions such as:

```
letrec
  even(x) = if zero?(x)
              then zero?(0)
              else (odd (x - 1))
  odd(x)  = if zero?(x)
              then zero?(1)
              else (even (x - 1))
in (odd 99)
```

Here is the result of parsing it:

```
AProg ([],
 Letrec
  ([("even", "x", None, None,
     ITE (IsZero (Var "x"), IsZero (Int 0),
      App (Var "odd", Sub (Var "x", Int 1))));
    ("odd", "x", None, None,
     ITE (IsZero (Var "x"), IsZero (Int 1),
      App (Var "even", Sub (Var "x", Int 1))))],
  App (Var "odd", Int 99)))
```

Note that `Letrec` now has two arguments:

```
type expr =
  | Var of string
  | Int of int
  | Add of expr*expr
  | Sub of expr*expr
  | Mul of expr*expr
  | Div of expr*expr
  | Let of string*expr*expr
  | IsZero of expr
  | ITE of expr*expr*expr
  | Proc of string*expr
  | App of expr*expr
  | Letrec of rdecs*expr
  | Record of (string*expr) list
  | Proj of expr*string
  | Cons of expr*expr
  | Hd of expr
  | Tl of expr
  | Empty of expr
  | EmptyList
  | Unit
  | Debug of expr
and
   rdecs = (string * string * texpr option * texpr option * expr) list
```

where `rdecs` is just a type synonym for a list of five-tuples. Thus, the first argument of `Letrec` is a list of five-tuples of the form (name of recursive function, name of parameter, ignore, ignore, body of the recursive function). See the parse tree above for an example.

4 Implementing **RECM**

A stub is provided for you. You must perform the following tasks, modifying the stub accordingly.

1. The `env` type in file `ds.ml` must be updated by modifying the `ExtendEnvRec` constructor from REC to hold a list of recursion closures:

```
type env =
  | EmptyEnv
  | ExtendEnv of string*exp_val*env
  | ExtendEnvRec of rdecs*env
```

   Note that in REC the constructor `ExtendEnvRec` was declared with an argument of type `string*string*expr*env`. In RECM it now supports a list of mutually recursive declarations (as indicated by the highlighted type).

2. Update `apply_env` in the file `ds.ml`. It currently reads as follows:

```
let rec apply_env : string -> exp_val ea_result =
  fun id env ->
  match env with
  | EmptyEnv -> Error (id^" not found!")
  | ExtendEnv(v,ev,tail) ->
    if id=v
```

```
    then Ok ev
    else apply_env id tail
  | ExtendEnvRec(v,par,body,tail) ->
    if id=v
    then Ok (ProcVal (par,body,env))
    else apply_env id tail
```

3. You will also have to update `interp.ml`:

```
...
| Letrec(decs, e2) ->
  error "implement"
```

   In fact, the code for `Letrec` you must produce will turn out to be very similar to that in REC. Any helper functions must be placed in `ds.ml`.

5 Trying Out Your Code

You may use `interp`, as usual:

```
utop # interp "
letrec
  even(x) = if zero?(x)
              then zero?(0)
              else (odd (x - 1))
  odd(x)  = if zero?(x)
              then zero?(1)
              else (even (x - 1))
in (odd 99)";;
- : exp_val Rec.Ds.result = Ok (BoolVal true)
```

utop

Alternatively you can type your code in a text file (which must be located in the `lib` folder), say `code.mrec`, and then use `interpf` instead of `interp`:

```
utop # interpf "code.mrec";;
- : exp_val Rec.Ds.result = Ok (BoolVal true)
```

utop

6 Submission instructions

Submit a file named `EC1.zip` through Canvas containing the entire stub with your updates to `interp.ml` and `ds.ml`. Make sure to type `dune clean` before creating the zip file. One submission per group. The name of the other member of your group must be posted as a canvas comment AND the names of both members of the team should appear as a comment at the top of the file `interp.ml`.