CS 496: Special Assignment 1
Due: 29 March, 11:55pm

March 23, 2020

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.

2 Assignment

This assignment consists in extending REC to allow for mutually recursive function definitions. The resulting language will be called REC-M. It modifies the concrete syntax for `letrec` as follows. The production

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

in REC is replaced with:

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

in REC-M. The expression { <Identifier>(<Identifier>) = <Expression>}$^+$ above means that there may be 1 or more declarations. Here is an example of a valid program in REC-M:

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

```
              else (even (x - 1))
in (odd 99)
```

Evaluating that expression should produce the result `NumVal 1`, 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 `NumVal 0` as a result. This is correct since 98 is not an odd number.

Note that the above expression is not syntactically valid in REC since it does not support mutually recursive definitions. To see this, try running it in the interpreter for REC(you will get a parse error).

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

```
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)
```

3 Implementing REC-M

To facilitate the process of implementing REC-M a stub has been provided for you in Canvas. This stub has been obtained by taking the interpreter for REC and applying some changes. Here is a summary of the changes:

1. The `parser.mly` file has been updated so that the parser is capable of parsing expressions such as

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

   Here is the result of parsing it:

   ```
   Letrec
    ([Dec ("even", "x",
       ITE (IsZero (Var "x"), Int 1, App (Var "odd", Sub (Var "x", Int 1))));
      Dec ("odd", "x",
       ITE (IsZero (Var "x"), Int 0, App (Var "even", Sub (Var "x", Int 1))))],
    App (Var "odd", Int 99))
   ```

   Note that `Letrec` (in file `ast.ml`) 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 decs*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
   decs = (string*string*expr) list
```

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

2. The `env` type has been updated by creating a new constructor to hold recursion closures:

```
type env =
  | EmptyEnv
  | ExtendEnv of string*exp_val*env
  | ExtendEnvRec of Ast.decs*env
```

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

You will have to update:

1. `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
```

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

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

In fact, the code for `LetrecEnv` should be very similar to that in REC. Note that this may require using helper functions that you would need to place in `ds.ml`.

4 Trying Out Your Code

We typically try out the interpreter by typing:

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

Alternatively you can type your code in a text file (located in the `src` folder) with a `rec` extension, say `code.rec`, and then use `interpf` instad of `interp`:

```
utop # interpf "code";;
- : exp_val Rec.Ds.result = Ok (NumVal 1)
```

The code is in the stub.

5 Submission instructions

Submit a file named `SA1_<SURNAME>.zip` through Canvas. Include all files from the stub. One submission per group. The name of the other member of your group must be posted as a canvas comment.