BPRD 2018

Vilfred Sikker Dreijer

December 2019

1 Assignment 1

1)

I've added 'Every' to make it print every number in the sequence

```
let iconEx1 = Every(Write(Prim("<", CstI 7, FromTo(1,10))))
```

2)

```
let iconEx2 = Every(Write(And(FromTo(1,4), And(Write (CstS "\n"),
    FromTo(1,4)))))
```

iconEx2 writes each Right for each Left. The nested 'And' creates a newline after FromTo(1,4) So each nested 'And' will be created for each of the first FromTo(1,4)

to create the following:

```
1 2 3 4
2 4 6 8
3 6 9 12
4 8 12 16 val it : value = Int 0
```

I changed iconEx2 to:

```
let iconEx2 = Every(Write(Prim("*", FromTo(1,4), And(Write( CstS "\
    n"), FromTo(1,4)))))
```

3)

My solution for the find method:

added to expr:

```
| Find of string * string
```

added to eval:

```
| Find(pat, str) ->
    let rec find (i:int) =
        let index = str.IndexOf(pat, i)
        if index <> -1
```

```
        then
            cont (Int index) (fun _ -> find (index+1))
        else
            econt ()
    find 0
```

4)

test strings and results

```
let str1 = "Hi there - if there are anyone"
> run (Every(Write(Find("there",str1))));;
3 14 val it : value = Int 0

let str2 = "this this is is stu stuttering"
> run (Every(Write(Find("this",str2))));;
0 5 val it : value = Int 0

let str3 = "hey you, yea you right there, no you!"
> run (Every(Write(Find("you",str3))));;
4 13 33 val it : value = Int 0
```

5)

Changed the run method to:

```
let str1 = "Hi there - if there are anyone"
> run (Every(Write(Prim("<", CstI 10, Find("e",str1)))));;
16 18 22 29 val it : value = Int 0
```

Assignment 2

1)

Convert to abstract syntax:

```
(* ex1 *)
let x = { } in x end (* ex1 *)
Let ("x", Record [], Var "x")

(* ex2 *)
let x = {field1 = 32} in x.field1 end
Let ("x",Record [("field1", CstI 32)],Field (Var "x","field1"))

(* ex3 *)
let x = {field1 = 32; field2 = 33} in x end
Let ("x", Record [("field1", CstI 32); ("field2", CstI 33)], Var "x
    ")

(* ex4 *)
let x = {field1 = 32; field2 = 33} in x.field1 end
Let ("x", Record [("field1", CstI 32); ("field2", CstI 33)], Field
    (Var "x", "field1"))

```

```
(* ex5 *)
let x = {field1 = 32; field2 = 33} in x.field1+x.field2 end
Let ("x", Record [("field1", CstI 32); ("field2", CstI 33)], Prim
    ("+", Field (Var "x", "field1"), (Var "x", "field2")))
```

2)

First I added the following to FunPar.fsy

```
%token LBRA RBRA DOT SEMI

Record:
    | NAME EQ Expr               { [($1, $3) ]    }
    | NAME EQ Expr SEMI Record  { ($1, $3) :: $5 }

AtExpr:
| LET NAME EQ LBRA RBRA IN Expr END        { Let($2, Record([]), $7
    ) }
| LET NAME EQ LBRA Record RBRA IN Expr End { Let($2, Record($5), $8
    ) }
```

The following was added to FunLex.fsl

```

| '.'             { DOT }
| ';'             { SEMI }
| '{'             { LBRA }
| '}'             { RBRA }
```

Results:

```
1)
> fromString "let x = { } in x end (* ex1 *)";;
val it : Absyn.expr = Let ("x",Record [],Var "x")

2)
> fromString "let x = {field1 = 32} in x.field1 end (* ex2 *)";;
val it : Absyn.expr =
Let ("x",Record [("field1", CstI 32)],Field (Var "x","field1"))

3)
> fromString "let x = {field1 = 32; field2 = 33} in x end (* ex3 *)
    ";;
val it :Absyn.expr =
Let ("x",Record [("field1", CstI 32); ("field2", CstI 33)],Var "x")

4)
> fromString "let x = {field1 = 32; field2 = 33} in x.field1 end (*
     ex4 *)";;
val it : Absyn.expr =
Let ("x",Record [("field1", CstI 32); ("field2", CstI 33)], Field (
    Var "x","field1"))

5)
> fromString "let x = {field1 = 32; field2 = 33} in x.field1+x.
    field2 end (* ex5 *)";;
val it : Absyn.expr =
```

```
23 Let ("x",Record [("field1", CstI 32); ("field2", CstI 33)], Prim
       ("+",Field (Var "x","field1"),Field (Var "x","field2")))
```

Assignment 3

2)

First I added the following to value in HigherFun.fs:

```
1 | RecordV of (string * value) list
```

The Result of run

```
1 > open HigherFun;;
2 > run (Let ("x",Record [("field1", CstI 32);("field1",CstI 33)],
      Field (Var "x","field1")));;
3 val it : value = Int 32
```