CSE 130 - Programming Assignment #4

OCaml

125 points

Notes and Hints:

Lexer and Parser , Interpreter . OCamlLex Tutorial OCamlYacc Tutorial (see submission instructions below)

(click your browser's refresh button to ensure that you have the most recent version)

(Programming Assignment #4 FAQ)

Note: See this for instructions on starting OCaml in the ACS lab machines. To download and install OCaml version 3.12.0 on your home machines see the instructions here. Remember that this is only to enable you to play with the assignment at home: the final version turned in must work on the ACS Linux machines. While you can use windows to begin working with OCaml, the code you turn in must be that required for the ACS Linux environment.

Code Documentation and General Requirements

Code for all programming assignments should be **well documented**. A working program with no comments will **receive only partial credit**. Documentation entails writing a description of each function/method, class/structure, as well as comments throughout the code to explain the program logic. Comments in OCaml are enclosed within (* *), and may be nested. It is understood that some of the exercises in this programming assignment require extremely little code and will not require extensive comments.

While few programming assignments pretend to mimic the "real" world, they may, nevertheless, contain some of the ambiguity that exists outside the classroom. If, for example, an assignment is amenable to differing interpretations, such that more than one algorithm may implement a correct solution to the assignment, it is incumbent upon the programmer to document not only the functionality of the algorithm (and more broadly his/her interpretation of the program requirements), but to articulate **clearly** the reasoning behind a particular choice of solution.

Assignment Overview

The overall objective of this assignment is to expose you to some advanced features of OCaml such as higher-order functions, abstract datatypes and modules, as well as to fully understand the notion of scoping, binding, environments and closures, by implementing a mini-ML interpreter. In addition, in this assignment you will be building a lexer and a parser. Again, no individual function requires more than 15-25 lines, so if you're answer is longer, you can be sure that you need to rethink your solution. The template for the assignment, as well as several test cases is available as a zip file pa4.zip that you need to download.

Note: All the solutions can be done using the purely functional fragment of OCaml, using constructs covered in class, and most require the use of *recursion*. Solutions using imperative features such as references or while loops will receive **no credit**. Feel free to use any library functions that you want.

It is a good idea to start this assignment early as it is somewhat longer than the first three assignments.

Even though we have included sample inputs and expected outputs for each step in the assignment, these tests *are not comprehensive*. It is possible to pass them all and still have a lot of mistakes (which our grading will expose). You are responsible for understanding the problems and testing other cases to make sure your implementation is correct.

Assignment Testing and Evaluation

Your functions/programs **must** compile and/or run on a **Linux** ACS machine (e.g. `ieng6.ucsd.edu` , as this is where the verification of your solutions will occur. While you may develop your code on any system, ensure that your code runs as expected on an ACS machine prior to submission. You should test your code in the directories from which the zip files (see below) will be created, as this will approximate the environment used for grading the assignment.

Most of the points, except those for comments and style, will be **awarded automatically, by evaluating your functions against a given test suite**. In the `tests` directory, there is a small handful of tests. For this project, you will use `make` to compile your program. If for some problem, you cannot get the code to compile, leave it as is with the `raise ...`, with your partial solution enclosed below as a comment. **There will be no exceptions to this rule**. If the code you submit does not compile with make, you will receive 0 for the assignment.

Submission Instructions

1. Create the zip file for submission

Your solutions to this assignment will be stored in separate files under a directory called `solution/`, inside which you will place the files : `config.make,main.ml,Makefile,nanoLex.mll,nano.ml,nanoParse.mly,rules.make,test.ml,toplevel.ml` and there should be **no other files in the directory**.

After creating and populating the directory as described above, create a zip file called `<LastName>_<FirstName>_cse130_pa4.zip` by going into the directory `solution` and executing the UNIX shell command:

```
zip <LastName>_<FirstName>_cse130_pa4.zip *
```

You can refer to an example submission file to compare with yours. Make sure that your zipped file's structure is the same as the example.

2. Test the zip file to check for its validity

Once you've created the zip file with your solutions, you will use the `validate_pa4` program to see whether your zip file's structure is well-formed to be inspected by our grading system by executing the UNIX shell command:

```
validate_pa4 <LastName>_<FirstName>_cse130_pa4.zip
```

The `validate_pa4` program will output `OK` if your zip file is well-formed and your solution is compiled. Otherwise, it will output some error messages. Before going to step 3, make sure that your zip file passes `validate_pa4` program. **Otherwise you get a zero for the whole assignment.** If you have any trouble with this, refer to the instructions in step 1.

3. Submit the zip file via the `turnin` program

Once your zip file passes the validation check by `validate_pa4`, you will use the `turnin_pa4` program to submit this file for grading by going into the directory `solution/` and executing the UNIX shell command:

```
turnin_pa4 <LastName>_<FirstName>_cse130_pa4.zip
```

The `turnin_pa4` program will provide you with a confirmation of the submission process; make sure that the size of the file indicated by `turnin_pa4` matches the size of your tar file. (turnin_pa4 is a thin wrapper script around the ACMS command `turnin` that repeats validation and ensures that the propper assignment name is passed) Note that you may submit multiple times, but your latest submission overwrites previous submissions, and will be the ONLY one we grade. If you submit before the assignment deadline, and again afterwards, we will count it as if you only submitted after the deadline. .

Data structures and overview (`nano.ml`)

```
type binop =
  Plus
| Minus
| Mul
| Div
| Eq      (* =  *)
| Ne      (* <> *)
| Lt      (* <  *)
| Le      (* <= *)
| And
| Or
| Cons

(* data types for Expressions in nano-ml *)
type expr =
  Const of int
| True
| False
| NilExpr
| Var of string
| Bin of expr * binop * expr
| If  of expr * expr * expr
| Let of string * expr * expr     (* let X = E1 in E2 -> Let (X,E1,E2) *)
| App of expr * expr              (* F X -> App(F,X) -- (calling function F w/ argument X) *)
| Fun of string * expr            (* fun X -> E -> Fun(X,E) *)
| Letrec of string * expr * expr  (* let rec X = E1 in E2 -> Letrec (X,E1,E2) *)

(* data types for Values in nano-ml *)
type value =
  Int of int
| Bool of bool
| Closure of env * string option * string * expr (* Closure(environment,Some name_of_function (* or None if anonymous *),formal,body) *)
| Nil
| Pair of value * value

and env = (string * value) list
```

WARNING: Before getting started please go through the following notes/tutorials:

- Lexer and Parser
- Interpreter
- OCamlLex Tutorial
- OCamlYacc Tutorial

Problem #1: ML-nano parser & lexer (`nanoParse.mly, nanoLex.mll`)

The goal of this problem is to write a parser and lexer for nano-ml using mlyacc.

In each subproblem, we will increase the complexity of the expressions parsed by your program.

(a)10 points

We will begin by making our parser recognize some of the simplest ML expressions: constants and variables.

Begin with `nanoParse.mly` and define tokens `TRUE`, `FALSE`, and `Id` (note that a token `Num` is already defined). An `Id` token should have a single agrument of a string, which will store the name of the variable.

Next add rules to `nanoLex.mll`. A `Num` constant is a sequence of one or more digits. An `Id` is a letter (capital or lowercase) followed by zero or more letters or digits. The strings "true" and "false" should return the corresponding tokens.

Finally, add a rule to `nanoLex.mll` to ignore whitespace: space, newline (\n), carriage return (\r), and tab (\t)

Once you have implemented this functionality, you should get the following behavior (`$` is a shell prompt, `#` is a nanoml.top prompt)

```
$ make
output from make, hopefully with no errors
$ ./nanoml.top
NanoML

$ ./nanoml.top
Objective Caml version 3.10.0

# NanoLex.token (Lexing.from_string "true");;
- : NanoParse.token = NanoParse.TRUE
# Main.token_list_of_string "true false 12345 foo bar baz";;
- : NanoParse.token list = [NanoParse.TRUE; NanoParse.FALSE; NanoParse.Num 12345; NanoParse.Id "foo"; NanoParse.Id "bar";
NanoParse.Id "baz"]
```

Now return to `nanoParse.mly`. Add rules to the parser so that true, false, integers, and ids are parsed into expressions (of type `Nano.expr` from `nano.ml`).

Once you have implemented this functionality, you should get the following behavior (`$` is a shell prompt, `#` is a nanoml.top prompt)

```
$ make
output from make, hopefully with no errors
$ ./nanoml.top
NanoML

$ ./nanoml.top
Objective Caml version 3.10.0

# NanoParse.exp NanoLex.token (Lexing.from_string "true");;
- : Nano.expr = Nano.True
# NanoParse.exp NanoLex.token (Lexing.from_string "false");;
- : Nano.expr = Nano.False
# NanoParse.exp NanoLex.token (Lexing.from_string " \n123");;
- : Nano.expr = Nano.Const 123
# NanoParse.exp NanoLex.token (Lexing.from_string "\rfoo");;
- : Nano.expr = Nano.Var "foo"
```

(b) 10 points

Add the following tokens to the lexer and parser.

String	Token
let	LET
rec	REC
=	EQ
in	IN
fun	FUN
->	ARROW
if	IF
then	THEN
else	ELSE

These should be parsed as in real ML to give `Nano.Let`, `Nano.Letrec`, `Nano.Fun`, and `Nano.If` expressions (of type `Nano.expr`). That is, a let expression should be "let <id> = <expr> in <expr>", a letrec expression should be the same, but with "rec" added. A fun expression should be "fun <id> -> <expr>", and an if expression should be "if <expr> then <expr> else <expr>". Here <id> denotes any id from part (a), and <expr> denotes any expression from part (a), or any let / letrec / fun / if expression.

Once you have implemented this functionality and recompiled, you should get the following behavior at the `./nanoml.top` prompt:

```
# Main.token_list_of_string "let rec foo = fun x -> if y then z else w in foo";;
```

```
- : NanoParse.token list = [NanoParse.LET; NanoParse.REC; NanoParse.Id "foo"; NanoParse.EQ; NanoParse.FUN; NanoParse.Id "x";
NanoParse.ARROW; NanoParse.IF; NanoParse.Id "y"; NanoParse.THEN; NanoParse.Id "z"; NanoParse.ELSE; NanoParse.Id "w";
NanoParse.IN; NanoParse.Id "foo"]
# Main.string_to_expr "let rec foo = fun x -> if y then z else w in foo";;
- : Nano.expr = Nano.Letrec ("foo", Nano.Fun ("x", Nano.If (Nano.Var "y", Nano.Var "z", Nano.Var "w")), Nano.Var "foo")
```

(c) 10 points

Add the following tokens to the lexer and parser.

String	Token
+	PLUS
-	MINUS
*	MUL
/	DIV
<	LT
<=	LE
!=	NE
&&	AND
||	OR

Add all of these as binary operators to your parser. Each should result in a `Nano.Bin` with the corresponding `Nano.binop`. The arguments to these binary operators may be any expressions. (You don't need to worry about types. "3+true||7" is allowed as far as the parser is concerned.)

Once you have implemented this functionality and recompiled, you should get the following behavior at the `./nanoml.top` prompt:

```
# Main.token_list_of_string "+ - /*|| <<= &&!=";;
- : NanoParse.token list = [NanoParse.PLUS; NanoParse.MINUS; NanoParse.DIV; NanoParse.MUL; NanoParse.OR; NanoParse.LT;
NanoParse.LE; NanoParse.AND; NanoParse.NE]
# Main.string_to_expr "x + y";;
- : Nano.expr = Nano.Bin (Nano.Var "x", Nano.Plus, Nano.Var "y")
# Main.string_to_expr "if x < 4 then a || b else a && b";;
- : Nano.expr = Nano.If (Nano.Bin (Nano.Var "x", Nano.Lt, Nano.Const 4),
Nano.Bin (Nano.Var "a", Nano.Or, Nano.Var "b"),
Nano.Bin (Nano.Var "a", Nano.And, Nano.Var "b"))
# Main.string_to_expr "if 4 <= z then 1-z else 4*z";;
- : Nano.expr = Nano.If (Nano.Bin (Nano.Const 4, Nano.Le, Nano.Var "z"),
Nano.Bin (Nano.Const 1, Nano.Minus, Nano.Var "z"),
Nano.Bin (Nano.Const 4, Nano.Mul, Nano.Var "z"))
# Main.string_to_expr "let a = 6 / 2 in a!=11";;
- : Nano.expr = Nano.Let ("a", Nano.Bin (Nano.Const 6, Nano.Div, Nano.Const 2),
Nano.Bin (Nano.Var "a", Nano.Ne, Nano.Const 11))
```

(d) 5 points

Add the following tokens to the lexer and parser.

String	Token
(	LPAREN
)	RPAREN

Add rules to your parser to allow parenthesized expressions. In addition, add a rule to your parser for function application. Function application is simply "<expr$_1$> <expr$_2$>", which corresponds to calling <expr$_1$> on argument <expr$_2$>.

Once you have implemented this functionality and recompiled, you should get the following behavior at the `./nanoml.top` prompt:

```
# Main.token_list_of_string "() ( )";;
- : NanoParse.token list = [NanoParse.LPAREN; NanoParse.RPAREN; NanoParse.LPAREN; NanoParse.RPAREN]
# Main.string_to_expr "f x";;
- : Nano.expr = Nano.App (Nano.Var "f", Nano.Var "x")
# Main.string_to_expr "(fun x -> x+x) (3*3)";;
- : Nano.expr = Nano.App (Nano.Fun ("x", Nano.Bin (Nano.Var "x", Nano.Plus, Nano.Var "x")),
Nano.Bin (Nano.Const 3, Nano.Mul, Nano.Const 3))
# Main.string_to_expr "(((add3 (x)) y) z)";;
- : Nano.expr = Nano.App (Nano.App (Nano.App (Nano.Var "add3", Nano.Var "x"), Nano.Var "y"), Nano.Var "z")
# Main.filename_to_expr "tests/t1.ml";;
- : Nano.expr = Nano.Bin (Nano.Bin (Nano.Const 2, Nano.Plus, Nano.Const 3), Nano.Mul,
Nano.Bin (Nano.Const 4, Nano.Plus, Nano.Const 5))
# Main.filename_to_expr "tests/t2.ml";;
- : Nano.expr = Nano.Let ("z1", Nano.Const 4, Nano.Let ("z", Nano.Const 3,
Nano.Let ("y", Nano.Const 2, Nano.Let ("x", Nano.Const 1, Nano.Let ("z1", Nano.Const 0,
Nano.Bin (Nano.Bin (Nano.Var "x", Nano.Plus, Nano.Var "y"), Nano.Minus, Nano.Bin (Nano.Var "z", Nano.Plus, Nano.Var "z1")))))))
```

(d) 20 points

Restructure your parser to give binary operators the following precedence and associativity. This will likely require that you add additional rules to your parser.

Operators	Associativity
High Precedence	
function application	left
`*`, `/`	left
`+`, `-`	left
`=`, `!=`, `<`, `<=`	left
`&&`	left
`||`	left
`let`, `fun`, `if`	N/A
Low Precedence	

Left associative means that "1-2-3-4" should be parsed as if it were "((1-2)-3)-4", and "f x y z" should be parsed as if it were "((f x) y) z".

Function application having higher precedence than multiplications, and multiplication higher than addition means that "1+f x*3" should be parsed as if it were "1+((f x)*3)"

Once you have implemented this functionality and recompiled, you should get the following behavior at the `./nanoml.top` prompt:

```
# Main.string_to_expr "1-2-3-4";;
- : Nano.expr = Nano.Bin (Nano.Bin (Nano.Bin (Nano.Const 1, Nano.Minus, Nano.Const 2), Nano.Minus, Nano.Const 3), Nano.Minus,
Nano.Const 4)
# Main.string_to_expr "1+a&&b||c+d*e-f/g x";;
- : Nano.expr = Nano.Bin (Nano.Bin (Nano.Bin (Nano.Const 1, Nano.Plus, Nano.Var "a"), Nano.And, Nano.Var "b"), Nano.Or, Nano.Bin
(Nano.Bin (Nano.Var "c", Nano.Plus, Nano.Bin (Nano.Var "d", Nano.Mul, Nano.Var "e")), Nano.Minus, Nano.Bin (Nano.Var "f",
Nano.Div, Nano.App (Nano.Var "g", Nano.Var "x"))))
# Main.filename_to_expr "tests/t13.ml";;
- : Nano.expr = Nano.Let ("f", Nano.Fun ("x", Nano.Fun ("y", Nano.Fun ("a",
Nano.Bin (Nano.App (Nano.Var "a", Nano.Var "x"), Nano.Mul, Nano.Var "y")))),
Nano.Let ("g", Nano.Fun ("x", Nano.Bin (Nano.Var "x", Nano.Plus, Nano.Bin (Nano.Const 1, Nano.Mul, Nano.Const 3))),
Nano.App (Nano.App (Nano.App (Nano.Var "f", Nano.Const 7), Nano.Const 8), Nano.Var "g")))
```

(e) 10 points extra credit

Add the following tokens to the lexer and parser.

String	Token
`[`	`LBRAC`
`]`	`RBRAC`
`;`	`SEMI`
`::`	`COLONCOLON`

Add rules to your lexer and parser to support parsing lists. "[a;b;c;d;e;f;g]" should be parsed as if it were "a::b::c::d::e::f::g::[]". The :: operator should have higher priority than the comparison functions (=, <=, etc.), and lower priority than + and -. In addition, :: should be right associative. [] should give `NilExpr`, and :: should be treated as any other binary operator.

Once you have implemented this functionality and recompiled, you should get the following behavior at the `./nanoml.top` prompt:

```
# Main.string_to_expr "1::3::5::[]";;
- : Nano.expr = Nano.Bin (Nano.Const 1, Nano.Cons, Nano.Bin (Nano.Const 3, Nano.Cons, Nano.Bin (Nano.Const 5, Nano.Cons,
Nano.NilExpr)))
# Main.string_to_expr "[1;3;5]";;
- : Nano.expr = Nano.Bin (Nano.Const 1, Nano.Cons, Nano.Bin (Nano.Const 3, Nano.Cons, Nano.Bin (Nano.Const 5, Nano.Cons,
Nano.NilExpr)))
# Main.string_to_expr "1::3::5::[]=[1;3;5]";;
- : Nano.expr =
Nano.Bin (Nano.Bin (Nano.Const 1, Nano.Cons, Nano.Bin (Nano.Const 3, Nano.Cons, Nano.Bin (Nano.Const 5, Nano.Cons,
Nano.NilExpr))),
Nano.Eq, Nano.Bin (Nano.Const 1, Nano.Cons, Nano.Bin (Nano.Const 3, Nano.Cons, Nano.Bin (Nano.Const 5, Nano.Cons,
Nano.NilExpr))))
```

Problem #2: ML-nano interpreter (`nano.ml`)

In this problem, you will implement an interpreter for a small fragment of ML.

(a) 15 points

```
type binop = Plus | Minus | Mul | Div

type expr = Const of int
    | Var of string
    | Bin of expr * binop * expr

type value = Int of int
```

```
type env = (string * value) list
```

First consider the types described above: `binop, expr` are used to capture simple ML expressions. Each expression is either an integer constant, a variable, or a binary operator applied to two sub-expressions. A value is an integer, and an environment is a list of pairs of variable names and values. Use `listAssoc` to write a function `lookup: string * env -> value` that finds the most recent binding for a variable (i.e. the first from the left) in the list representing the environment. Using this function, write a function `eval : env * expr -> value` that when called with the pair `(evn,e)` evaluates an ML-nano expression `e` of the above type, in the environment `evn` , and raises an exception `MLFailure ("variable not bound: x")` if the expression contains an unbound variable.

Once you have implemented this functionality and recompiled, you should get the following behavior at the `./nanoml.top` prompt:

```
# open Nano;;
# let evn =[("z1",Int 0);("x",Int 1);("y",Int 2);("z",Int 3);("z1",Int 4)];;
val evn : (string * Nano.value) list = [("z1", Int 0); ("x", Int 1); ("y", Int 2); ("z", Int 3); ("z1", Int 4)]
# let e1 =Bin(Bin(Var "x",Plus,Var "y"), Minus, Bin(Var "z",Plus,Var "z1"));;
val e1 : Nano.expr = Bin (Bin (Var "x", Plus, Var "y"), Minus, Bin (Var "z", Plus, Var "z1"))
# eval (evn,e1);;
- : Nano.value = Int 0
# eval (evn,Var "p");;
Exception: Nano.MLFailure "Variable not bound: p".
```

(b) 15 points

```
type binop = Plus | Minus | Mul | Div
           | Eq | Ne | Lt | Le | And | Or

type expr = Const of int
          | True | False
          | Var of string
          | Bin of expr * binop * expr
          | If  of expr * expr * expr

type value = Int of int
           | Bool of bool

type env = (string * value) list
```

Add support for the binary operators `=`, `!=`, `<`, `<=`, `&&`, `||`. This will require using the new value type `Bool`. The operators `=` and `!=` should work if both operands are `Int` values, or if both operands are `Bool` values. The operators `<` and `<=` are only defined for `Int` arguments, and `&&` and `||` are only defined for `Bool`arguments. For all other arguments, a MLFailure exception should be raised with an appropriate error message.

Now implement `If` expressions. Given `If(p,t,f)`, `p` should be evaluated first, and if it evaluates to true (as a `Bool`), then `t` should be evaluated, and the value of the if expression should be the value of `t`. Similarly, if `p` evaluates to false, then `f` should be evaluated and the result returned. If `p` does not evaluate to a `Bool`, a MLFailure exception should be raised with an appropriate error message.

Once you have implemented this functionality and recompiled, you should get the following behavior at the `./nanoml.top` prompt:

```
# open Nano;;
# let evn =[("z1",Int 0);("x",Int 1);("y",Int 2);("z",Int 3);("z1",Int 4)];;
val evn : (string * Nano.value) list = [("z1", Int 0); ("x", Int 1); ("y", Int 2); ("z", Int 3); ("z1", Int 4)]
# let e1 =If(Bin(Var "z1",Lt,Var "x"),Bin(Var "y",Ne,Var "z"),False);;
val e1 : Nano.expr = If (Bin (Var "z1", Lt, Var "x"), Bin (Var "y", Ne, Var "z"), False)
# eval (evn,e1);;
- : Nano.value = Bool true
# let e2 =If(Bin(Var "z1",Eq,Var "x"),Bin(Var "y",Le,Var "z"),Bin(Var "z",Le,Var "y"));;
val e2 : Nano.expr = If (Bin (Var "z1", Eq, Var "x"), Bin (Var "y", Le, Var "z"), Bin (Var "z", Le, Var "y")) # eval (evn,e2);;
- : Nano.value = Bool false
```

(c) 10 points

```
type expr = ...
    | Let of string * expr * expr
    | Letrec of string * expr * expr
```

Now consider the extended the types as shown above to include "let-in" expressions that introduce local bindings exactly as in OCaml. `Let (b,e1,e2)` should be evaluated as the ML expression `let b = e1 in e2` . Similarly, `Letrec (b,e1,e2)` should be evaluated as `let rec b = e1 in e2` . (Since at this point, we do not support functions, Let and Letrec should do the same thing.)

Once you have implemented this functionality and recompiled, you should get the following behavior at the `./nanoml.top` prompt:

```
# open Nano;;
# let e1 = Bin(Var "x",Plus,Var "y");;
val e1 : Nano.expr = Bin (Var "x", Plus, Var "y")
# let e2 = Let("x",Const 1,Let("y",Const 2,e1));;
val e2 : Nano.expr = Let ("x", Const 1, Let ("y", Const 2, Bin (Var "x", Plus, Var "y")))
# eval ([],e2);;
- : Nano.value = Int 3
# let e3 = Let("x",Const 1,Let("y",Const 2,Let("z",e1,Let("x",Bin(Var "x",Plus,Var "z"),e1))));;
val e3 : Nano.expr = Let ("x", Const 1, Let ("y", Const 2, Let ("z", Bin (Var "x", Plus, Var "y"), Let ("x", Bin (Var "x", Plus,
Var "z"), Bin (Var "x", Plus, Var "y")))))
# eval ([],e3);;
```

```
- : Nano.value = Int 6
```

(d) 15 points

```
type expr = ...
            | App of expr * expr
            | Fun of string * expr

type value = ...
              | Closure of env * string option * string * expr
```

We now extend the above types so that there is an expression for function application: `App(e1,e2)` corresponds to applying `e2` to the function `e1` , we allow function declarations via the expression `Fun (x,e)` where `x` and `e` are respectively the formal parameter and body-expression of the function. For now, assume the function is not recursive. However, functions do have values represented by the `Closure (evn,f,x,e)` where `evn` is the environment at the point where that function was declared, and `n,x,e` are name, formal and body expression of the function. If the function is anonymous or declared in a let statement, the name should be `None`. If the function is declared in a `let rec` statement, then the name of the function should be `Some f` (where f is the name of the function). Extend your implementation of `eval` by adding the appropriate cases for the new type constructors.

Once you have implemented this functionality and recompiled, you should get the following behavior at the `./nanoml.top` prompt:

```
# open Nano;;
# eval ([],Fun ("x",Bin(Var "x",Plus,Var "x")));;
- : Nano.value = Closure ([], None, "x", Bin (Var "x", Plus, Var "x"))
# eval ([],App(Fun ("x",Bin(Var "x",Plus,Var "x")),Const 3));;
- : Nano.value = Int 6
# let e3=Let("h",Fun("y",Bin(Var "x", Plus, Var "y")),App(Var "f",Var "h"));;
val e3 : Nano.expr = Let ("h", Fun ("y", Bin (Var "x", Plus, Var "y")), App (Var "f", Var "h"))
# let e2 = Let("x",Const 100,e3);;
val e2 : Nano.expr = Let ("x", Const 100, Let ("h", Fun ("y", Bin (Var "x", Plus, Var "y")), App (Var "f", Var "h")))
# let e1 = Let("f",Fun("g",Let("x",Const 0,App(Var "g",Const 2))),e2);;
val e1 : Nano.expr =
Let ("f", Fun ("g", Let ("x", Const 0, App (Var "g", Const 2))),
Let ("x", Const 100,
Let ("h", Fun ("y", Bin (Var "x", Plus, Var "y")),
App (Var "f", Var "h"))))
# eval ([],e1);;
- : Nano.value = Int 102
# eval ([],Letrec("f",Fun("x",Const 0),Var "f"));;
- : Nano.value = Closure ([], Some "f", "x", Const 0)
```

(e) 15 points

Make the above work for recursively defined functions (when declared with `let rec`).

Once you have implemented this functionality and recompiled, you should get the following behavior at the `./nanoml.top` prompt:

```
# open Nano;;
# eval ([],Letrec("fac",Fun("n",If(Bin(Var "n",Eq,Const 0),Const 1,Bin(Var "n",Mul,App(Var "fac",Bin(Var "n",Minus,Const
1))))),App(Var "fac",Const 10)));;
- : Nano.value = Int 3628800
```

(f) 10 points extra credit

```
   type binop = ...
              | Cons

type expr = ...
          | NilExpr

type value = ...
           | Nil
           | Pair of value * value
```

Extend your program to support operations on lists. In addition to the changes to the data types, add support for two functions "hd" and "tl" which do what the corresponding ML functions do.

Once you have implemented this functionality and recompiled, you should get the following behavior at the `./nanoml.top` prompt:

```
# open Nano;;
# eval ([],Bin(Const 1,Cons,Bin(Const 2,Cons,NilExpr)));;
- : Nano.value = Pair (Int 1, Pair (Int 2, Nil))
# eval ([],App(Var "hd",Bin(Const 1,Cons,Bin(Const 2,Cons,NilExpr))));;
- : Nano.value = Int 1
# eval ([],App(Var "tl",Bin(Const 1,Cons,Bin(Const 2,Cons,NilExpr))));;
- : Nano.value = Pair (Int 2, Nil)
```

Problem #3: ML-nano executable

Once you have completed all the above parts, you should end up with an executable "nanoml.byte". You should be able to test it as follows from the shell prompt:

```
$ ./nanoml.byte tests/t1.ml
...
out: 45
$ ./nanoml.byte tests/t2.ml
...
out: 0
$ ./nanoml.byte tests/t3.ml
...
out: 2
```

and so forth, for all the files in tests. To get the expected value for the other tests, run them with ocaml:

```
# #use "tests/t1.ml";;
- : int = 45
```

and so forth. "tests/t14.ml" requires that you have completed both extra credit parts.

If the code you submit does not get passed the test with `validate_pa4`, you will get 0 for the assignment.