PRACTICE REPORT

Implementation of a Lambda Interpreter Calculus in Ocaml

Authors
Antonio Vila Leis
(antonio.vila@udc.es)

Óscar Alejandro Manteiga Seoane
(oscar.manteiga@udc.es)

Index

User's manual

Compilation and execution

To get started with the interpreter, the first thing to do is to compile the code. From a terminal in the file path, we will do a "make all" to generate all the necessary files. There are other functionalities available in Makefile as "make clean", which will remove the files resulting from the compilation (lexer.ml, parser.mli, parser.ml, top, *.cmi, *.cmo, *~).

Once this is done, we just have to run it with "./top" (or with rlwrap ./top) to access the program. To pass all the examples provided in examples.txt, we will have to run it with "./top < examples.txt".

Functionalities

Multi-line expressions

The interpreter supports expressions that are on separate lines. For this reason, every time you need to finish one, you have to write ";;" on the same line. If we separate these two characters in multiple lines, it will not work (providing a lexical error), in the same way that happens in Ocaml.

An example of this functionality could be:

```
if true
then 1
else 0
;;
- : Nat = 1

if true then 1 else 0;;
- : Nat = 1
```

As you can see, we can create an if then else expression on several lines and the program will work in the same way as if it were written on a single line.

Internal fixed-point combinator

An important feature in the interpreter is the incorporation of functions created by direct recursive definitions. This makes it easier to declare functions due to avoid them becoming so long and tedious, being able to write a function in a direct way without having to define several functions that we will use in a general one.

An example of this functionality could be to go from having to write the following function:

```
let fix = lambda f.(lambda x. f (lambda y. x x y)) (lambda x. f
(lambda y. x x y)) in let sumaux = lambda f. (lambda n. (lambda m. if
(iszero n) then m else succ (f (pred n) m))) in let sum = fix sumaux
in let prodaux = lambda f. (lambda n. (lambda m. if (iszero m) then 0
else sum n (f n (pred m)))) in let prod = fix prodaux in prod 10 5;;
- : Nat = 120
```

To only have to write:

```
letrec prod : Nat -> Nat -> Nat -> Nat =
  lambda n : Nat. lambda m : Nat.
    letrec sum : Nat -> Nat -> Nat -> Nat =
      lambda n : Nat. lambda m : Nat. if iszero n then m else succ
(sum (pred n) m)
    in
      if iszero n then 0 else sum (m) (prod (pred n) m)
  in
    prod 10 5;;
- : Nat = 120
```

Context of global definitions

Another interesting feature is the association of variable names to values or terms that can be used for later expressions. This way we will be able to use the program in a faster way, even though it increases the complexity of what we can do.

An example of this functionality could be:

```
x = {true, false};;
x : {Bool * Bool} = {true, false})
y = {"a", "b"};;
y : {String * String} = {"a", "b"}
z = {x, y};;
z : {{Bool * Bool} * {String * String * String}} = {{true, false},
{"a", "b"}}

x;;
- : {Bool * Bool} = {true, false}
y;;
- : {String * String} = {"a", "b"}
z;;
- : {{Bool * Bool} * {String * String * String}} = {{true, false},
{"a", "b"}}
z.1;;
- : {{Bool * Bool} = {true, false}, {"a", "b"}}
z.2;;
- : {String * String} = {"a", "b"}
```

As we can see, the program stores in x, y, z the corresponding tuples, being able to access them later and even being able to use those variables to create new expressions.

Strings

The interpreter has the String type, in charge of supporting character strings. To obtain an expression of this type, just type the desired string between " ". It can also be used in any operation previously available.

In addition to the incorporation of this new type, we also added the concatenation operation, responsible for joining String strings into a single string. An example of this functionality could be:

```
"I am a String type";;
- String = "I am a String type";;

concat "I am a " "String type";;
- : String = "I am a String type"
```

Pairs/tuples

The interpreter supports the use of pairs (two-element tuples) and tuples of any size. In this way, different types of values can be stored within the same element. In addition, the operation projection returns the value of the element n that we indicate to it in the form pair/tuple.n. First and second operations have not been included due to the implementation of the projections, because they are redundant.

An example of this functionality could be:

```
tupla1 = {true, 1, {"asd", false}};;
tupla1 : {Bool, Nat, {String, Bool}} = {true, 1, {"asd", false}}

tupla1.1;;
- : Bool = true

tupla1.3;;
- : {String, Bool} = {"asd", false}

tupla1.3.2;;
- : Bool = false
```

Records

The interpreter supports the use of records of any type, even of different types from each other. In this way, different fields can be stored under labels within the same element, i.e. each field has its own label. Projection operations based on the labels are also available.

An example of this functionality could be:

```
registro1 = {a=true, b=1, c={"asd", {z=true, x=false}}};;
registro1 : {a:Bool, b:Nat, c:{String, {z:Bool, x:Bool}}} = {a=true,
b=1, c={"asd", {z=true, x=false}}}

registro1.c;;
- : {String, {z:Bool, x:Bool}} = {"asd", "fgh", {z=true, x=false}}

registro1.c.2.z;;
- : Bool = true
```

Lists

The interpreter supports the use of lists of any type, but exclusively of one type. In this way, it is possible to store different types of values associated with a label within the same element. This new type has the operations of obtaining head (hd), tail (tl) and query if it is empty (isempty).

An example of this functionality could be:

```
empty = nil [Nat];;
empty : List[Nat] = []

one = cons [Nat] 1 empty;;
one : List[Nat] = 1::[]

two = cons [Nat] 2 one;;
two : List[Nat] = 2::1::[]

head [Nat] two;;
- : Nat = true

tail [Nat] two;;
- : List[Nat] = true

isnil [Nat] empty;;
- : Bool = true

isnil [Nat] one;;
- : Bool = false
```

Unit

As last functionality, we have the Unit type, its only possible value is (), it is used when we want to specify that a function does not have a specific return value, but simply performs an action. This allows us to write sequences like unit; unit; unit; term. This ensures that t1; t2 is equivalent to lambda x: Unit. t2 by evaluating the last term. If t1 is not type Unit it would give an error, otherwise we evaluate t2 and provide the result.

An example of this functionality would be:

```
unit;;
- : Unit = unit

unit; unit; unit; true;;
- : Bool = true
```

Technical Manual

Explanation of each file

File main.ml

In the main file we have the read_line_semicolon function, which reads a user input line and returns a string containing all lines read until a ';' character is encountered.

Next, the function top_level_loop is defined, which is a loop that is executed repeatedly until the user enters an end-of-file (EOF). Within the loop, a prompt line is printed to request an input from the user, and the read_line_semicolon function is called to read the input. Then, the Parser and Lexer module is used to parse the input and convert it to a valid lambda expression. Finally, the execute function is called with the lambda expression and the variable and type contexts to evaluate the expression and obtain a result.

In case an error occurs during parsing or evaluation, Lexical_error, Parse_error, Type_error exceptions are handled, and an appropriate error message is printed on the screen. When an end-of-file is detected, a goodbye message is printed and the loop terminates. Finally, top_level_loop is called to start the execution of the interpreter.

File lexer.mll

In this code, a token function associated with the Parser module is defined. The token function is used to parse an input character string and convert it into a token recognized by the lambda expression interpreter. The token function is defined as a parsing rule using the rule command.

The parsing rule specifies different input patterns that can be recognized by the token function, and assigns each pattern a specific token. For example, if the keyword "lambda" or "L" is encountered, the token LAMBDA is returned, if the keyword "true" or "false" is encountered, the corresponding token TRUE or FALSE is returned.

Different types of special tokens are also handled, such as parentheses, semicolons, and equals and colons. In addition, integers and strings are recognized, and the corresponding INTV and STRINGV tokens are returned. If an end-of-file (EOF) is encountered, the EOF token is returned. If none of the patterns match the input, a Lexical_error exception is thrown.

File parser.mly

This code defines a parser for the lambda expression interpreter in OCaml. The parser is implemented by using the OCaml Yacc library, which is a parser generator for programming languages.

The file starts by declaring some necessary libraries with the open command, and defines a Lexical_error exception from the Parser module. It then defines a set of tokens that can be recognized by the parser, using the %token command. These tokens correspond to the keywords, operators and special symbols that are part of the lambda expression language.

Special tokens are also defined for INTV integers, STRING strings, and STRINGV variable names. These tokens are used to recognize numeric values, character strings, and variable names in lambda expressions.

Next, the initial language symbol is defined, which is the starting point of the parser, and the type of data returned by the parser is set. In this case, the initial symbol is s and the data type returned is a Lambda.command.

Finally, the grammar of the lambda expression language is defined by parsing rules. Each rule specifies a valid input pattern, and assigns to that pattern an action that is executed when the pattern is recognized in the input. The action can be any valid OCaml code that uses the tokens recognized in the input pattern to construct an appropriate data structure (in our case, lambda.ml and lambda.mli).

File lambda.ml y lambda.mli

This code defines a lambda expression interpreter in OCaml. It begins by defining the ty data type, which represents the different types of data that can appear in lambda expressions. The data types include TyBool for Boolean values, TyNat for natural values, TyArr for functions that take one type and return another, TyString for character strings, TyTuple for value tuples, TyRecord for value records, and TyUnit for unit value.

The term data type is also defined, which represents the different lambda expressions that can be evaluated by the interpreter. Lambda expressions include Boolean constants, arithmetic operations, conditionals, variable declarations, functions and function applications, operations with strings, tuples and records, and other types of expressions.

The code also defines the command data type, which represents the commands that can be executed by the interpreter. Commands include Eval to evaluate a lambda expression, and Bind to assign a lambda expression to a variable.

In addition, the code defines the data type 'a context, which represents a context of variables that can be used in lambda expressions. The context is a list of pairs of variable names and values associated with those variables.

We also define a number of functions that can be used to work with the defined data types and commands. For example, the emptyctx function returns an empty context, the addbinding function adds a new variable to a given context, and the getbinding function gets the value of a given variable in a given context.

The string_of_ty function converts a data type to a string, the Type_error exception is thrown when a type error is encountered in a lambda expression, and the typeof function determines the type of given lambda expression in a given type context.

The string_of_term function converts a lambda expression to a string, the NoRuleApplies exception is thrown when there are no rules that apply to a given lambda expression during its evaluation, and the eval function evaluates a lambda expression in a given variable context.

Finally, the execute function executes a given command in a given variable context and a given type context, and returns the new variable context and the new type context after command execution.

File Makefile

The Makefile provides a way to automate the compilation of an OCaml project. When the "make" command is run in the terminal, Makefile rules are executed that compile the files lambda.ml, parser.mly, lexer.mll and main.ml and then link these

compiled files into an executable called "top". The "clean" rule removes intermediate and compiled files generated during compilation.

File examples.txt

Finally, this last file contains examples and tests that can be sent to the top executable (./top < examples.txt) to check that the elements evaluated in the interpreter work.

Changes for each function

Multi-line expressions

To address this functionality, we made several changes to the main.ml. At first, we thought that we have had to modify also the lexer to find the characters ";", but later we realized that it was enough to make a function in the main. The first modification was to introduce read_line_semicolon(), which searches for a ";" to stop reading the expression and concatenate everything that has been read.

```
let read_line_semicolon() =
 let rec read_aux linea =
   let linea2 = read_line() in
     if (contains linea2 ';') then
       let pos = (rindex_from linea2 ((length linea2) - 1) ';') in
         if (pos = 0) then
           read_aux (linea)
         else
           if (linea2.[(pos - 1)] = ';') then (linea ^ (sub linea2 0 (pos - 1)))
           else read_aux (linea ^ (sub linea2 0 (pos - 1)) ^ " ")
     else read_aux (linea ^ linea2 ^ " ")
 in read_aux "";;
```

Later we made a minor change so that it would only stop if it found two consecutive semicolons. We had to change this in order to implement the Unit type, leaving the function above.

Internal fixed point combiner

In order to implement this functionality, the changes made start in the lexer.ml, adding the recognition of the word letrec:

```
rule token = parse
  | "letrec"     { LETREC }
```

For the parser.mly file what was done was to add the new token, change Lambda.type to Lambda.command, modify the start term and add to term the recognition of the letrec operation:

```
%token LETREC
%type <Lambda.command> s
```

```
s :
  term EOF
    { Eval $1 }

term :
  | LETREC STRINGV COLON ty EQ term IN term
      { TmLetIn ($2, TmFix (TmAbs ($2, $4, $6)), $8) }
```

Finally, for lambda.ml and lambda.mli, the Fix operation is added to the corresponding functions, the command function is included and the final part of evaluation and execution is changed:

```
type term =
 | TmFix of term
;;

type command =
  Eval of term
  | Bind of string * term

let rec typeof tctx tm = match tm with
  | TmFix t1 ->
      let tyT1 = typeof tctx t1 in
      (match tyT1 with
      TyArr (tyT11, tyT12)->
      if tyT11 = tyT12 then tyT12
      else raise (Type_error "result of body not compatible with domain")
      | _ -> raise (Type_error "arrow type expected"))
;;

let rec string_of_term = function
  | TmFix t ->
      "(fix " ^ string_of_term t ^ ")"
;;

let rec free_vars tm = match tm with
  | TmFix t ->
      free_vars t
;;

let rec subst x s tm = match tm with
  | TmFix t ->
      TmFix (subst x s t)
;;
```

```
let rec eval1 vctx tm  = match tm with
    (* E-FixBeta *)
  | TmFix (TmAbs (x, _, t2)) ->
      subst x tm t2

    (*E-Fix*)
  | TmFix t1 ->
      let t1' = eval1 vctx t1 in
      TmFix t1'
    (*E-Var*)
;;

let apply_ctx ctx tm =
  List.fold_left (fun t x -> subst x (getbinding ctx x) t) tm (free_vars tm)

let rec eval vctx tm =
  try
      let tm' = eval1 vctx tm in
      eval vctx tm'
  with
      NoRuleApplies -> apply_ctx vctx tm
;;

let execute (vctx, tctx) = function
  Eval tm ->
      let tyTm = typeof tctx tm in
      let tm' = eval vctx tm in
      print_endline ("- : " ^ string_of_ty tyTm ^ " = " ^ string_of_term tm');
      (vctx, tctx)
  | Bind (s, tm) ->
      let tyTm = typeof tctx tm in
      let tm' = eval vctx tm in
      print_endline ( s ^ " : " ^ string_of_ty tyTm ^ " = " ^ string_of_term
tm');
      (addbinding vctx s tm', addbinding tctx s tyTm)
```

Context of global definitions

In this case, the modifications start in parser.mly, with only a change in the initial term:

```
s :
   STRINGV EQ term EOF
     { Bind ($1, $3) }
```

For lambda-ml and lambda.mli the changes focus on the TmVar operation and slight changes with respect to the previous section:

```
type 'a context =
(string * 'a) list
```

```
let rec eval1 vctx tm  = match tm with
  | TmVar s ->
      getbinding vctx s
;;

let apply_ctx ctx tm =
  List.fold_left (fun t x -> subst x (getbinding ctx x) t) tm (free_vars tm)

let rec eval vctx tm =
  try
      let tm' = eval1 vctx tm in
      eval vctx tm'
  with
      NoRuleApplies -> apply_ctx vctx tm
;;

let execute (vctx, tctx) = function
  Eval tm ->
      let tyTm = typeof tctx tm in
      let tm' = eval vctx tm in
      print_endline ("- : " ^ string_of_ty tyTm ^ " = " ^ string_of_term tm');
      (vctx, tctx)
  | Bind (s, tm) ->
      let tyTm = typeof tctx tm in
      let tm' = eval vctx tm in
      print_endline ( s ^ " : " ^ string_of_ty tyTm ^ " = " ^ string_of_term
tm');
      (addbinding vctx s tm', addbinding tctx s tyTm)
```

Strings

To realize the String type, the first thing that was done was the modification of the lexer.mll file. This was done in order to recognize the new operation, the new type and to be able to collect the String value in a variable.

```
rule token = parse
  | "concat"    { CONCAT }
 | "String"    { STRING }
 | '"'[^'"' ';' '\n']*'"'          { STRV (String.sub (Lexing.lexeme
lexbuf) 1 ((String.length (Lexing.lexeme lexbuf))-2))}
```

The next thing that was changed was the parser.mly, where we wanted the concat operation to be recognized, the TmString could be called from recognizing a String value and that the TyString itself existed.

```
%{
 open Lambda;;
%}
```

```
%token CONCAT
%token STRING
%token <string> STRV

appTerm :
 | CONCAT pathTerm pathTerm
     { TmConcat ($2, $3) }

atomicTerm :
 | STRV
     { TmString $1 }

atomicTy :
 | STRING
     { TyString }
```

Finally, new lines were added to lambda.ml and lambda.mli to add the corresponding operations. First, the String type and the terms for concat and String were added. These terms are the ones that allow to do the operations of typeof, eval... These functions were also modified to add the desired behavior. In the typeof function, for example, what we perform for the concat is the verification that both values passed from the parser are of type tyString. The lines we modified are the following:

```
(* TYPE DEFINITIONS *)
type ty =
  | TyString
;;

type term =
  | TmString of string
  | TmConcat of term * term
;;

(* TYPE MANAGEMENT (TYPING) *)
let rec string_of_ty ty = match ty with (*Para declarar tipos nuevos*)
  | TyString ->
      "String"
;;

let rec typeof tctx tm = match tm with (* Reglas de tipado *)
    (* T-String *)
  | TmString _ ->
      TyString
```

```
    (* T-Concat *)
  | TmConcat (t1, t2) ->
      let tyT1 = typeof tctx t1 in
      let tyT2 = typeof tctx t2 in
      if tyT1 = TyString && tyT2 = TyString then TyString
      else raise (Type_error "arguments of concat are not both
strings")
;;

(* TERMS MANAGEMENT (EVALUATION) *)
let rec free_vars tm = match tm with
  | TmString _ ->
      []
  | TmConcat (t1, t2) ->
      lunion (free_vars t1) (free_vars t2)
;;

let rec subst x s tm = match tm with
  | TmString _ ->
      tm
  | TmConcat (t1, t2) ->
      TmConcat (subst x s t1, subst x s t2)
;;

let rec isval tm = match tm with
  | TmString _ -> true
;;

let rec eval1 vctx tm = match tm with
      (*E-ConcatString*)
  | TmConcat (TmString s1, TmString s2) ->
      TmString (s1 ^ s2)

  | TmConcat (TmString s1, t2) ->
      let t2' = eval1 vctx t2 in
      TmConcat (TmString s1, t2')

      (*E-Concat*)
  | TmConcat (t1, t2) ->
      let t1' = eval1 vctx t1 in
      TmConcat (t1', t2)
;;
```

```
let rec string_of_term = function
  | TmString s ->
      "\"" ^ s ^ "\""
  | TmConcat (t1, t2) ->
      "concat " ^ "(" ^ string_of_term t1 ^ ", " ^ string_of_term t2 ^
")"
;;
```

Pairs/Tuples and Records

To realize the Tuples (since the type Pairs was not raised as it is a 2-element tuple) and the Records, the first thing to be modified was the lexer.mll. What was done was to add the key recognition ("{", "}") to be able to create these structures:

```
rule token = parse
 | '{'          { LCURLY }
 | '}'          { RCURLY }
```

Next, the parser.mly was changed, adding the necessary tokens, setting the projection operation in the pathTerm (it can be a dot followed by a number for tuples or a dot followed by a string for records), including in the atomicTerm the format of each one and calling auxiliary functions to do them recursively (relying on Ocaml lists). Finally, we did the same in the atomicTy:

```
%{
 open Lambda;;
%}

%token LCURLY
%token RCURLY

pathTerm :
    pathTerm DOT STRINGV
      { TmProj ($1, $3) }
  | pathTerm DOT INTV
      { TmProj ($1, string_of_int $3) }

atomicTerm :
  | LCURLY tupleFields RCURLY
      { TmTuple $2 }
  | LCURLY recordFields RCURLY
       { TmRecord $2 }

tupleFields :
      term
      { [$1] }
  | term COMMA tupleFields
      { $1 :: $3 }
recordFields :
```

```
      /* Empty */
      { [] }
  | notEmptyRecordFields
       { $1 }

notEmptyRecordFields :
      notEmptyRecordField
      { [$1] }
  | notEmptyRecordField COMMA notEmptyRecordFields
      { $1 :: $3 }

notEmptyRecordField :
      STRINGV EQ term
      { ($1, $3) }

atomicTy :
  | LCURLY tupleFieldTypes RCURLY
      { TyTuple $2 }
  | LCURLY recordFieldTypes RCURLY
      { TyRecord $2 }

tupleFieldTypes :
      ty
      { [$1] }
  | ty COMMA tupleFieldTypes
      { $1 :: $3 }

recordFieldTypes :
      /* empty */
      { [] }
  | notEmptyRecordFieldTypes
      { $1 }

notEmptyRecordFieldTypes :
      notEmptyRecordFieldType
      { [$1] }
  | notEmptyRecordFieldType COMMA notEmptyRecordFieldTypes
      { $1 :: $3 }

notEmptyRecordFieldType :
      STRINGV COLON ty
      { ($1, $3) }
```

Finally, it remains to edit the lambda.ml and lambda.mli. As in String, we modify the types, terms and required operations. For the operations, the most important is the projection, which has as main elements the typeof (to check types) and eval1 (where the evaluation of the same is developed to cast the result):

```
type ty =
  | TyTuple of ty list
```

```
  | TyRecord  of (string * ty) list
;;

type term =
  | TmTuple of term list
  | TmRecord of (string * term) list
  | TmProj of term * string;;

let rec string_of_ty ty = match ty with
  | TyTuple tys ->
      "{" ^ String.concat ", " (List.map string_of_ty tys) ^ "}"
  | TyRecord fields ->
      "{" ^ String.concat ", " (List.map (fun (l, ty) -> l ^ ":" ^
string_of_ty ty) fields) ^ "}";;

let rec typeof tctx tm = match tm with
    (* T-Tuple *)
  | TmTuple fields ->
      TyTuple (List.map (fun t -> typeof tctx t) fields)

    (* T-Record *)
  | TmRecord fields ->
      TyRecord (List.map (fun (s, t) -> (s, typeof tctx t)) fields)

      (* T-Proj *)
  | TmProj (t, s) ->
      (match typeof tctx t with
         TyRecord fieldtys -> (try List.assoc s fieldtys with
            Not_found -> raise (Type_error ("label "^s^" not found")))
       | TyTuple fieldtys ->
         (try List.nth fieldtys (int_of_string s - 1) with
           _ -> raise (Type_error ("label " ^ s ^ " not found ")))
       | _ -> raise (Type_error "tuple or record type expected"))
;;

let rec free_vars tm = match tm with
  | TmTuple fields ->
      List.fold_left (fun fv ti -> lunion (free_vars ti) fv) [] fields
  | TmRecord fields ->
      List.fold_left (fun fv (lb, ti) -> lunion (free_vars ti) fv) []
fields
  | TmProj (t, lb) ->
      free_vars t
;;
```

```
let rec subst x s tm = match tm with
  | TmTuple fields ->
      TmTuple (List.map (fun ti -> subst x s ti) fields)
  | TmRecord fields ->
      TmRecord (List.map (fun (lb, ti) -> (lb, subst x s ti)) fields)
  | TmProj (t, lb) ->
      TmProj (subst x s t, lb)
;;

let rec isval tm = match tm with
  | TmTuple fields -> List.for_all (fun ti -> isval ti) fields
  | TmRecord fields -> List.for_all (fun (lb, ti) -> isval ti) fields
;;

let rec eval1 vctx tm = match tm with
   (*E-Tuple*)
  | TmTuple fields ->
      let rec eval_fields fields = match fields with
            [] -> raise NoRuleApplies
      | t::ts when isval t -> let ts' = eval_fields ts in t::ts'
      | t::ts -> let t' = eval1 vctx t in t'::ts
      in
      TmTuple (eval_fields fields)

      (*E-Record*)
  | TmRecord fields ->
      let rec eval_fields fields = match fields with
            [] -> raise NoRuleApplies
      | (l,t)::ts when isval t -> let ts' = eval_fields ts in
(l,t)::ts'
      | (l,t)::ts -> let t' = eval1 vctx t in (l,t')::ts
      in
      TmRecord (eval_fields fields)

      (* E-ProjTuple *)
  | TmProj (TmTuple fields as v1, lb) when isval v1 ->
      List.nth fields (int_of_string lb - 1)

      (* E-ProjRcd *)
  | TmProj (TmRecord fields as v1, lb) when isval v1 ->
      List.assoc lb fields
```

```
  (* E-Proj *)
  | TmProj (t1, lb) ->
     let t1' = eval1 vctx t1 in
     TmProj (t1', lb)
;;

let rec string_of_term = function
  | TmTuple fields ->
     "{" ^ String.concat ", " (List.map string_of_term fields) ^ "}"
  | TmRecord fields ->
     "{" ^ String.concat ", " (List.map (fun (s, t) -> s ^ "=" ^
string_of_term t) fields) ^ "}"
  | TmProj (t, s) ->
     string_of_term t ^ "." ^ s
;;
```

Lists

To make the Lists, the first thing to be modified was the lexer.mll, adding the recognition of braces ("[", "]"), the type "List" and the terms "nil", "cons", "isnil", "head" and "tail" in order to create these structures:

```
rule token = parse
  | "nil"       { NIL }
  | "cons"      { CONS }
  | "isnil"     { ISNIL }
  | "head"      { HEAD }
  | "tail"      { TAIL }
  | "List"      { LIST }
  | '['         { LSQUARE }
  | ']'         { RSQUARE }
```

Next, the parser.mly was changed, adding the necessary tokens, setting the cons, isnil, head and tail operation in the appTerm, including in the atomicTerm the format of the list and calling ty that will search in atomicTy for the type of values in the list:

```
%token NIL
%token CONS
%token ISNIL
%token HEAD
%token TAIL

%token LIST

%token LSQUARE
%token RSQUARE
```

```
appTerm :
  | CONS LSQUARE ty RSQUARE pathTerm pathTerm
      { TmCons ($3, $5, $6) }
  | ISNIL LSQUARE ty RSQUARE pathTerm
      { TmIsNil ($3, $5) }
  | HEAD LSQUARE ty RSQUARE pathTerm
      { TmHead ($3, $5) }
  | TAIL LSQUARE ty RSQUARE pathTerm
      { TmTail ($3, $5) }

atomicTerm :
  | NIL LSQUARE ty RSQUARE
      { TmNil $3 }

atomicTy :
  | LIST LSQUARE ty RSQUARE
      { TyList $3 }
```

Finally, new lines were added to lambda.ml and lambda.mli. Operations such as TmNil, TmCons, ,TmIsNil, TmHead and TmTail were incorporated. These are included in the typeof, free_vars, subst, eval1 and string_of_term functions. For the types, we added TyLists, which will be the basis for building them. This was also defined in string_of_ty.

```
type ty =
 | TyList of ty
;;

type term =
 | TmNil of ty
 | TmCons of ty * term * term
 | TmIsNil of ty * term
 | TmHead of ty * term
 | TmTail of ty * term
;;

let rec string_of_ty ty = match ty with
 | TyList ty ->
     "List[" ^ string_of_ty ty ^ "]"
;;

let rec typeof tctx tm = match tm with
   (* T-Nil *)
 | TmNil tyT ->
     TyList tyT
```

```
   (* T-Cons *)
 | TmCons (tyT, t1, t2) ->
     if typeof tctx t1 = tyT then
       if typeof tctx t2 = TyList tyT then TyList tyT
       else raise (Type_error "second argument of cons is not a list")
     else raise (Type_error "first argument of cons does not have this
list's element type")

   (* T-IsNil *)
 | TmIsNil (tyT, t1) ->
     if typeof tctx t1 = TyList tyT then TyBool
     else raise (Type_error "argument of isnil is not a list")

   (* T-Head *)
 | TmHead (tyT, t1) ->
     if typeof tctx t1 = TyList tyT then tyT
     else raise (Type_error "argument of head is not a list")

   (* T-Tail *)
 | TmTail (tyT, t1) ->
     if typeof tctx t1 = TyList tyT then TyList tyT
     else raise (Type_error "argument of tail is not a list")

   (* T-Unit *)
 | TmUnit ->
     TyUnit
;;

let rec free_vars tm = match tm with
 | TmNil tyT ->
   []
 | TmCons (tyT, t1, t2) ->
   lunion (free_vars t1) (free_vars t2)
 | TmIsNil (tyT, t1) ->
   free_vars t1
 | TmHead (tyT, t1) ->
   free_vars t1
 | TmTail (tyT, t1) ->
   free_vars t1
;;
```

```
let rec subst x s tm = match tm with
 | TmNil tyT ->
     TmNil tyT
 | TmCons (tyT, t1, t2) ->
     TmCons (tyT, subst x s t1, subst x s t2)
 | TmIsNil (tyT, t1) ->
     TmIsNil (tyT, subst x s t1)
 | TmHead (tyT, t1) ->
     TmHead (tyT, subst x s t1)
 | TmTail (tyT, t1) ->
     TmTail (tyT, subst x s t1)
;;

let rec isval tm = match tm with
 | TmNil _ -> true
 | TmCons (_, v1, v2) -> isval v1 && isval v2
;;

let rec eval1 vctx tm = match tm with
   (* E-IsNil *)
 | TmIsNil (tyT, t1) ->
     let t1' = eval1 vctx t1 in
       if t1' = TmNil tyT then
         TmTrue
       else
         TmFalse

   (* E-Cons *)
 | TmCons (tyT, t1, t2) ->
     let t1' = eval1 vctx t1 in
     if isval t1' then
       TmCons (tyT, t1', t2)
     else
       raise (Type_error "TmCons error")

   (* E-ConsHead *)
 | TmHead (_, TmCons(_, v1, _)) when isval v1 ->
     v1

   (* E-Head *)
 | TmHead (tyT, t1) ->
     let t1' = eval1 vctx t1 in
       TmHead(tyT, t1')
```

```
   (* E-ConsTail *)
 | TmTail (_, TmCons(_, _, v2)) when isval v2 ->
     v2

   (* E-Tail *)
 | TmTail (tyT, t1) ->
     let t1' = eval1 vctx t1 in
       TmTail(tyT, t1')

 | _ -> raise NoRuleApplies;;

let rec string_of_term = function
 | TmNil t1 ->
     "[]"
 | TmCons (tyT, t1, t2) ->
     string_of_term t1 ^ "::" ^ string_of_term t2
 | TmHead (t1, t2) ->
     "head " ^ "(" ^ string_of_term t2 ^ ")"
 | TmTail (t1, t2) ->
     "tail " ^ "(" ^ string_of_term t2 ^ ")"
 | TmIsNil (t1, t2) ->
     "isnil " ^ "(" ^ string_of_term t2 ^ ")";;
```

Unit

To create the Unit type, the lexer.mll file was first modified, adding this type as "Unit" associated to UNIT and its value as "unit" associated to UNITV. The lines added are shown below:

```
rule token = parse
  | "unit"      { UNITV }
  | "Unit"      { UNIT }
```

The parser.mly file was also tweaked, adding the necessary tokens (UNIT and UNITV), the UNITV type (which calls TmUnit) and the rule to accept a sequence of semicolon-separated expressions in term, and the UNIT (associated to TyUnit) in atomicTy. This can be seen in the following lines:

```
%token UNITV
%token UNIT

term :
  | UNITV
    { TmUnit }
  | term SEMICOLON term
     { TmApp (TmAbs ("x", TyUnit, $3), $1) }
```

Finally, new lines were added to lambda.ml and lambda.mli to add the corresponding operations. First, the Unit type and its term were added. Finally, it was added to already implemented functionalities, namely free_vars, subst, isval and string_of_term.

```
type ty =
  | TyUnit
;;

type term =
  | TmUnit
;;

let rec string_of_ty ty = match ty with
  | TyUnit ->
      "Unit"
;;

let rec typeof tctx tm = match tm with
    (* T-Unit *)
  | TmUnit ->
      TyUnit
;;

let rec free_vars tm = match tm with
  | TmUnit ->
      []
;;

let rec subst x s tm = match tm with
  | TmUnit ->
      TmUnit
;;

let rec isval tm = match tm with
  | TmUnit -> true
;;

let rec eval1 vctx tm = match tm with
  (* E-Unit *)
  | TmUnit ->
      TmUnit
;;
```