Memory

Alejandro Dopico Castro: alejandro.dopico2@udc.es

Abel Juncal Suárez: abel.juncal@udc.es

User Manual

1. Improvement in introduction and lambda expressions writing:

1.1. Multiline expressions recognition:

Now, you can employ more than one line to insert an input. You can write and then press Enter many times as wanted until the input doesn't have two consecutive semicolons (“;;”):

In next section, you can see examples of this functionality since are written with multiline.

2. Lambda-calculus language expansion:

2.1. Internal fixed point combiner incorporation:

Examples:

Prod:

```
>> letrec sum : Nat -> Nat -> Nat =
    lambda n: Nat. lambda m: Nat. if iszero n then m else succ (sum (pred n) m)
    in letrec prod : Nat -> Nat -> Nat =
    L m:Nat. L n:Nat. if iszero m then 0 else sum n ( prod (pred m) n)
        in prod 5 10;;
- .: Nat = 50
```

Fibonacci:

```
>> letrec sum : Nat -> Nat -> Nat =
   lambda n:Nat. lambda m: Nat. if iszero n then m else succ(sum(pred n) m)
       in letrec fib : Nat -> Nat =
       lambda x : Nat. if iszero x then 0 else if iszero(pred x) then 1 else sum (fib (pred(pred x))) (fib (pred x))
       in fib 10;;
- .: Nat = 55
```

Factorial:

```
>> letrec sum : Nat -> Nat -> Nat =
    lambda n: Nat. lambda m: Nat. if iszero n then m else succ (sum (pred n) m)
        in letrec prod : Nat -> Nat -> Nat =
        lambda m:Nat. lambda n:Nat. if iszero m then 0 else sum n ( prod (pred m) n)
            in letrec factorial : Nat -> Nat =
                lambda n: Nat. if iszero n then 1 else prod n (factorial (pred n))
                in factorial 5;;
- .: Nat = 120
```

2.2. Definition global context incorporation:

You can define variables of values or terms to use them later in lambda expressions. Our implementation is imperative since a value of a variable is changed, the result of an expression with that variable would change too.

2.3. Type String incorporation:

This new type includes also a concat operation. You can define a string as a sequence of characters between quotation marks.

Examples:

- “”
- “abc”
- concat “abc” “def”
    - This would produce the new string “abcdef”
- concat "abc” (concat “def” “ghi”)
    - This would produce the new string “abcdefghi”

2.4. & 2.5. Tuple and pairs incorporation:

You can define a tuple of elements with two braces (“{}”) and elements separated by commas (“,”). This can have several types inside. Inside can be all types of terms like another tuples or records, being nested.

{element1, element2, ...}

To get a tuple element you can use the projection like:

tuple_name.element_index

Examples:

- { 1, 2 }
    - {1, 2}.1 will return 1
- {“a”, 1, true}
- { {1, “b”}, { 2, “c”}}
    - { {1, “b”}, { 2, “c”}}.2 will return {2, “c”} as it’s the second element.
    - Even could be { {1, “b”}, { 2, “c”}}.2.1 and would get 2 as it’s the first element of the second element (tuple).

2.6. Record incorporation:

You can define a record with two braces (“{}”) and their elements with an id followed by an equal and the value, different elements separated by commas (“, “). As defined before there could be distinct types, even tuples or records nested.

{id1 = value1, id2 = value2, id3 = value3, …}

As mentioned before with tuple you can get a record element with a projection:

record_name.element_id

Examples:

- { x = 1, y = 2}
    - {x = 1, y = 2}.1 would return 1.
- {x = 1, y = true, z = {“a”, “b”}}

2.7. Lists incorporation:

Examples:

To define a list, we will use the terms cons (list constructor) and nil (empty list). A list must always end in nil to be correct. These terms are built with:

- An empty list: nil[type]
- A list constructor: cons[type] term term.
    - Second term can be another list constructor or the empty list.

- A list of nats with numbers from 1 to 3 would look like:
    - cons [Nat] 1 (cons [Nat] 2 (cons [Nat] 3 nil [Nat]))
- A list of other type like Bool would look:
    - cons [Bool] true (cons [Bool] false nil [Bool])
- A list of diverse types would cause a type error:
    - cons [Nat] 3 (cons [Bool] true nil [String])

Length:

```
>> letrec sum : Nat -> Nat -> Nat =
    lambda n: Nat. lambda m: Nat. if iszero n then m else succ (sum (pred n) m)
    in letrec length: List[Nat] -> Nat =
        lambda l : List[Nat]. if isnil[Nat] l then 0 else sum 1 (length (tail[Nat] l))
        in length cons[Nat] 1 (cons[Nat] 2 (cons[Nat] 3 nil[Nat]));;
- .: Nat = 3
```

Append:

```
>> letrec append: List[Nat] -> List[Nat] -> List[Nat] =
    lambda l1 : List[Nat]. lambda l2 : List[Nat]. if isnil[Nat] l1 then l2 else cons[Nat] (head[Nat] l1) (append (tail[Nat] l1) l2)
    in append (cons[Nat] 1 (cons[Nat] 2 (cons[Nat] 3 nil[Nat]))) (cons[Nat] 2 (cons[Nat] 3 (cons[Nat] 4 nil[Nat])));;
- .: Nat list = cons[Nat] 1 cons[Nat] 2 cons[Nat] 3 cons[Nat] 2 cons[Nat] 3 cons[Nat] 4 nil[Nat]
```

Map:

```
>> letrec map: (Nat -> Nat) -> List[Nat] -> List[Nat] =
    lambda f : Nat -> Nat. lambda l : List[Nat]. if isnil[Nat] l then nil[Nat] else cons[Nat] (f (head[Nat] l)) (map  f (tail[Nat] l))
    in map (lambda x: Nat.1) (cons[Nat] 1 (cons[Nat] 2 (cons[Nat] 3 nil[Nat])));;
- .: Nat list = cons[Nat] 1 cons[Nat] 1 cons[Nat] 1 nil[Nat]
```

Technical Manual

1. Improvement in introduction and lambda expressions writing:

1.1. Multiline expressions recognition:

To implement this improvement, we changed main.ml file, including two new functions:

- A check_line functions where we check if one line has two consecutive “;” characters.

```
let check_line line =
  try
    String.rindex_from line ((String.rindex line ';')-1)  ';'; true
  with
    Not_found -> false
```

-A recursive read_line_loop function, link the lines until the condition in first function is satisfied.

```
let rec read_line_loop string =
  let line = read_line() in
    if check_line line then String.sub (string ^ " " ^ line) 0 (String.length(string ^ " " ^ line)-2)  else string ^ " " ^ read_line_loop (line)
```

Finally, we change where the line was read using our second function.

```
let c = s token (from_string (read_line_loop "")) in
```

2. Lambda-calculus language expansion:

2.1. Internal fixed point combiner incorporation:

We had to define a new type Tmfix, so we start changing lambda.ml (We also include it in the interface, lambda.mli):

```
| TmFix of term
```

We included a new match in next functions:

-In typeof the typing rule T-Fix:

```
  (* T-Fix *)
| TmFix t1 ->
    let tyT1 = typeof ctx 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"))
```

-string_of_term which is the string of term:

```
| TmFix t ->
    "(fix " ^ string_of_term t ^ ")"
```

-free_vars where check for free vars in term t:

```
| TmFix t ->
    free_vars t
```

-subst where occurs the substitution of term t:

```
| TmFix t ->
    TmFix (subst x s t)
```

- In eval1 the evaluations rule E-FixBeta and E-Fix:

```
  (* E-FixBeta *)
| TmFix (TmAbs (x, _, t12)) ->
    subst x tm t12

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

In the parser.mly, we included in term another match:

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

2.2. Definition global context incorporation:

This one is more the more complex one, we add a new vcontext that it is a context composed by values making a few more changes in various files:

First, we change lambda.ml implementation

-We change the type context definition to include polymorphism:

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

-Now we will work with commands because we can receive from the input a term or a definition. So, we had to define type command:

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

-Function eval1 will receive the vcontext:

```
let rec eval1 vctx tm = match tm with
```

-In case of eval1, we had to add a case to the pattern matching:

```
| TmVar s ->
    getbinding vctx s
```

-In case of eval, we change the try where we call also a new function called apply_ctx. This function takes the value context and apply it in the term that we introduce:

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

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

-Finally, we added to this file a new function executes, that manages what context call after receiving a new “command”:

```
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)
```

After this, turn to change lambda.mli, where we change:

-type context to accept polymorphism and we define new type command

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

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

-We change the signature of all context functions:

```
val emptyctx : 'a context;;
val addbinding : 'a context -> string -> 'a -> 'a context;;
val getbinding : 'a context -> string -> 'a;;
```

-We also change typeof, eval and execute signature:

```
val typeof : ty context -> term -> ty;;
```

```
val eval : term context -> term -> term;;

val execute : term context * ty context -> command -> term context * ty context;;
```

In file main.ml:

Only had to change loop to receive a tuple of context:

```
let rec loop (vctx, tctx) =
```

Finally, we change parser.mli and parser.mly:

-In the first one, we modify vals to return a lambda.command not a lambda.term:

```
(Lexing.lexbuf  -> token) -> Lexing.lexbuf -> Lambda.command
```

-In the second one, we change “s” pattern matching, where we now look if we receive a declaration and not a term. We also define a new %type called lambda.command:

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

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

2.3. Type String incorporation:

In this case, we had to add a new type so we started modifying lambda.ml where we look for new matches in ty and term. We also had to add a concat operation, so we included it too but only in term (This step is replied in the lambda interface file, lamda.mli):

```
| TyString
```

```
| TmString of string
| TmConcat of term * term
```

We also, had to add match cases to string_of_ty, typeof, string_of_term, free_vars, subst and eval_1(only for concat). Evaluation rules must be invented but there are quite easy to understand:

-In String_of_ty:

```
| TyString  ->
    "String"
```

```
| TmString _  ->
    TyString
| TmConcat (t1, t2) ->
    if typeof ctx t1 = TyString && typeof ctx t2 = TyString then TyString
    else raise (Type_error "string type expected")
```

(typeof)

In concat case, we check two strings as terms of concatenation

```
| TmString s ->
    "\"" ^ s ^ "\""
| TmConcat (s1, s2) ->
    string_of_term s1 ^ string_of_term s2
```
(string_of_term)

```
| TmString _ ->
  []
| TmConcat (t1, t2) ->
  lunion (free_vars t1) (free_vars t2)
```
(free_vars)

```
| TmString s ->
    TmString s
| TmConcat (s1, s2) ->
    TmConcat (subst x s s1 ,subst x s s2)
```
(subst)

```
| TmConcat (TmString t1, TmString t2) ->
    TmString (t1 ^ t2)

| TmConcat (TmString t1, t2) ->
    let t2' = eval1 vctx t2 in
    TmConcat (TmString t1, t2')

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

Here we have 3 cases: where both are string, when one is string and the other is unknown and where two are unknown. In last two it evaluates unknown terms.

We also had to modify two more files: lexer.mll and parser.mly:

In the first one we had to include three more tokens: concat string and strv. First two are quite simple to understand but last one consists of define how a string can be received:

```
| "concat"    { CONCAT }
```

```
| "String"    { STRING }
```

```
| '"'[^ '"' ';' '\n']*'"'
                { let s = Lexing.lexeme lexbuf in
                STRV (String.sub s 1 (String.length s - 2))}
```

In case of parser.mly, here we include again the three new tokens defined in Lexer:

```
%token CONCAT
%token STRING
```

```
%token <string> STRV
```

We also include new matches in:

-app_term, where we look for concat:

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

-atomicTerm, where we look for a STRV entry:

```
| STRV {TmString $1}
```

-and atomicTy, where we look for a string

```
| STRING
    { TyString}
```

2.4. & 2.5. Tuple and pairs incorporation:

Here, we went directly to the tuple implementation, so we consider both sections as only one, since both are accepted.

The way to work with tuples is a little bit tricky. So, what we did is process it to a Ocaml List, so it ends with a tuple of one only element. This makes work with tuples much easier.

We have modified various files:

In lambda.ml:

We define the type and the term. We also defined projection type so we could retrieve the value of an element. Also did it in the interface (lambda.mli). I This would also be useful for the implementation of the records:

```
 | TyTuple of ty list
```

```
| TmTuple of term list
```

```
| TmProj of term * string
```

Like the String incorporation, we must implement some evaluation rules in this file:

-In string_of_ty, we apply it element by element:

```
| TyTuple ty ->
    let rec aux list = match list with
        h::[] -> string_of_ty h
      | h::t -> (string_of_ty h ^ ", ")  ^ aux t
      | [] -> raise (Invalid_argument "tuple cannot be empty")
    in "{" ^ aux ty ^ "}"
```

-In typeof we used the map function as it's a Ocaml List to implement T-Tuple and T-Proj typing rules:

```
| TmTuple t1 ->
    TyTuple (List.map (typeof ctx) t1)
```

```
| TmProj (t, s) ->
  (match typeof ctx t with
    TyTuple list -> (try List.nth list (int_of_string s - 1) with
      | _ -> raise (Type_error ("label " ^ s ^ " not found")))
```

-In string_of_term:

```
| TmTuple list ->
  let rec aux = function
      | [] -> ""
      | [h] -> string_of_term h
      | h::t -> string_of_term h ^ ", " ^ aux t
  in "(" ^ aux list ^ ")"
```

```
| TmProj (t, s) -> string_of_term t ^ "." ^ s
```

-In free_vars. The free vars of a tuple are the union of each element free variables:

```
| TmTuple t ->
    let rec aux list = match list with
        | h::[] -> free_vars h
        | h::t -> lunion (free_vars h) (aux t)
        | [] -> []
    in aux t
```

```
| TmProj (t, _) -> free_vars t
```

-In subst:

```
| TmTuple t ->
    TmTuple (List.map (subst x s) t)
```

```
| TmProj (t, lb) ->
    TmProj (subst x s t, lb)
```

-In isval:

```
| TmTuple list -> List.for_all (fun t -> isval t) list
```

-In eval1 we implemented E-Tuple, E-ProjTuple and E-Proj:

```
| TmTuple tuple ->
    let rec evalfield = function
      [] -> raise NoRuleApplies
    | vi::rest when isval vi ->
        let rest' = evalfield rest in vi::rest'
    | ti::rest ->
        let ti' = eval1 vctx ti in ti'::rest
    in let tuple' = evalfield tuple in TmTuple tuple'
```

```
| TmProj (TmTuple list as v, s) when isval v ->
  List.nth list (int_of_string s - 1)
```

```
 | TmProj (t, s) ->
     let t' = eval1 vctx t in
    TmProj (t', s)
```

In the lexer.mll, we include new token: “{“, “}” and “,”:

```
| ','           { COMMA }
| '{'           { LBRACE }
| '}'           { RBRACE }
```

In parser.mly, we define a new pathTerm between appTerm and atomicTerm to recognize the projection:

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

In atomic term we included new match. Under this we declared the tupleTerm that makes the transformation of the tuple of elements to a list having all elements:

```
| LBRACE tupleTerm RBRACE
    { TmTuple $2 }
```

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

2.6. Record incorporation:

This section was quite easy to implement, it's the same as the tuple ones, but instead of working with value elements it works with elements that have a tuple composed by an identifier and a value (string * term). No more explication exactly same changes that last one but changing this detail.

We define the type and the term. We also defined projection type so we could retrieve the value of an element. Also did it in the interface (lambda.mli). I This would also be useful for the implementation of the records:

```
  | TyRecord of (string * ty) list
  | TmRecord of (string * term) list
```

-In string_of_ty, we apply it element by element:

```
| TyRecord ty ->
  let rec aux list = match list with
      (i, h)::[] -> i ^ ":" ^ string_of_ty h
    | (i, h)::t -> (i ^ ":" ^ string_of_ty h ^ ", ")  ^ aux t
    | [] -> raise (Invalid_argument "record cannot be empty")
  in "{" ^ aux ty ^ "}"
```

-In typeof we used the map function as it's a Ocaml List to implement T-Rcd and T-Proj typing rules:

```
| TmRecord t1 ->
    TyRecord (List.combine (List.map fst t1) (List.map (typeof ctx) (List.map snd t1)))
```

-In string_of_term:

```
| TmRecord list ->
    let rec aux = function
      | [] -> ""
      | [(i, h)] -> i ^ " : " ^ string_of_term h
      | (i, h)::t -> i ^ " : " ^ string_of_term h ^ ", " ^ aux t
    in "(" ^ aux list ^ ")"
```

-In free_vars. The free vars of a record are the union of each element free variables:

```
| TmRecord t ->
  let rec aux list = match list with
      | (i, h)::[] -> free_vars h
      | (i, h)::t -> lunion (free_vars h) (aux t)
      | [] -> []
  in aux t
```

-In subst:

```
| TmRecord t ->
    TmRecord (List.combine (List.map fst t) (List.map (subst x s) (List.map snd t)))
```

-In isval:

```
| TmRecord list -> List.for_all (fun t -> isval t) (List.map snd list)
```

-In eval1 we implemented E-Rcd and E-ProjRcdj evaluation rules:

```
| TmRecord record ->
  let rec evalfield = function
    [] -> raise NoRuleApplies
  | (lb, vi)::rest when isval vi ->
      let rest' = evalfield rest in (lb, vi)::rest'
  | (lb, ti)::rest ->
      let ti' = eval1 vctx ti in (lb, ti')::rest
  in let record' = evalfield record in TmRecord record'
```

```
| TmProj (TmRecord list as v, s) when isval v ->
  List.assoc s list
```

In parser.mly:

-In atomic term we included new match. Under this we declared the recordTerm that makes the transformation of the record of elements to a list having all elements:

```
| LBRACE recordTerm RBRACE
    { TmRecord $2 }
```

```
recordTerm :
    STRINGV COLON term
        { [($1, $3)]}
  | STRINGV COLON term COMMA recordTerm
        { ($1, $3)::$5 }
```

2.7. Lists incorporation:

Lists are of only one type as all elements must be the same type.

So, we start defining the type for list and term for the list operations in the lambda.ml and lambda.mli:

```
| TyList of ty
```

```
| TmNil of ty
| TmCons of ty * term * term
| TmIsnil of ty * term
| TmHead of ty * term
| TmTail of ty * term
```

Then in string_of_ty we added a new match, since is only a type is quite simple.

```
| TyList ty ->
    string_of_ty ty ^ " list"
```

Now it is time to typeof, looking for the typing rules T-Nil, T-Cons, T-Isnil, T-Head and T-Tail:

```
| TmNil t ->
    TyList t

  (* T-Cons *)
| TmCons (ty, t1, t2) ->
    let tyT1 = typeof ctx t1 in
    let tyT2 = typeof ctx t2 in
    (match tyT2 with
        TyList ty21 ->
          if ty21 = tyT1 then TyList ty
          else raise (Type_error "Types of list don't match")
      | _ -> raise (Type_error "List type expected")
    )

  (* T-Isnil *)
| TmIsnil (ty, t1) ->
    let tyT1 = typeof ctx t1 in
    (match tyT1 with
        TyList _ -> TyBool
      | _ -> raise (Type_error "Type list expected")
    )

  (* T-Head*)
| TmHead (ty, t1) ->
    let tyT1 = typeof ctx t1 in
    (match tyT1 with
        TyList _ -> ty
      | _ -> raise (Type_error "Type list expected"))

  (* T-Tail *)
| TmTail (ty, t1) ->
    let tyT1 = typeof ctx t1 in
    (match tyT1 with
        TyList ty -> TyList ty
      | _ -> raise (Type_error "Type list expected")
    )
```

In string_of_term, we make the string with the term, type and content.

```
| TmNil t -> "nil[" ^ string_of_ty t ^ "]"
| TmCons (ty, t1, t2) ->
    "cons[" ^ string_of_ty ty ^ "] " ^ string_of_term t1 ^ " " ^ string_of_term t2
| TmIsnil (ty, t) ->
    "IsNil " ^ string_of_ty ty ^ " " ^ string_of_term t
| TmHead (ty, t) ->
    "Head " ^ string_of_ty ty ^ " " ^ string_of_term t
| TmTail (ty, t) ->
    "Tail " ^ string_of_ty ty ^ " " ^ string_of_term t
```

-free_vars:

```
| TmNil _ ->
    []
| TmCons (ty, t1, t2) ->
    lunion (free_vars t1) (free_vars t2)
| TmIsnil (ty, t) ->
    free_vars t
| TmHead (ty, t) ->
    free_vars t
| TmTail (ty, t) ->
    free_vars t
```

-subst where do the substitution of term t, this term can be a list constructor or an empty list:

```
| TmNil t ->
    TmNil t
| TmCons (ty, t1, t2) ->
    TmCons (ty, (subst x s t1), (subst x s t2))
| TmIsnil (ty, t) ->
    TmIsnil (ty, (subst x s t))
| TmHead (ty, t) ->
    TmHead (ty, (subst x s t))
| TmTail (ty, t) ->
    TmTail (ty, (subst x s t))
```

-In isval, when an empty list is received always is true and in a constructor when the first and second term are:

```
| TmNil _ -> true
| TmCons (_, t1, t2) -> isval t1 && isval t2
```

-eval1 where we implemented all the corresponding evaluation rules:

```
  (* E-Cons2*)
| TmCons (ty, v1, t2) when isval v1 ->
    let t2' = eval1 vctx t2 in
    TmCons (ty, v1, t2')

  (* E-Cons1*)
| TmCons (ty, t1, t2) ->
    let t1' = eval1 vctx t1 in
    TmCons (ty, t1', t2)

  (* E-IsnilNil*)
| TmIsnil (_, (TmNil _)) ->
    TmTrue

  (* E-IsnilCons*)
| TmIsnil (_, TmCons (_, v1, v2)) when (isval v1 && isval v2) ->
    TmFalse

  (* E-Isnil*)
| TmIsnil (ty, t1) ->
    let t1' = eval1 vctx t1 in
    TmIsnil (ty, t1')

  (* E-HeadCons *)
| TmHead (_, TmCons (_, v1, v2)) when (isval v1 && isval v2) ->
    v1

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

  (* E-TailCons *)
| TmTail (_, TmCons (_, v1, v2)) when (isval v1 && isval v2) ->
    v2

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

In file lexer.mll we included new tokens to define a list and terms:

```
| "nil"        { NIL }
| "cons"       { CONS }
| "isnil"      { ISNIL }
| "head"       { HEAD }
| "tail"       { TAIL }
```

In parser.mly we also included new tokens:

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

And new matches to atomicTerm:

```
| NIL LBRACKET ty RBRACKET
    { TmNil $3 }
| CONS LBRACKET ty RBRACKET atomicTerm atomicTerm
    { TmCons ($3, $5, $6) }
| ISNIL LBRACKET ty RBRACKET atomicTerm
    { TmIsnil ($3, $5) }
| HEAD LBRACKET ty RBRACKET atomicTerm
    { TmHead ($3, $5) }
| TAIL LBRACKET ty RBRACKET atomicTerm
    { TmTail ($3, $5) }
```

And atomicTy:

```
  | LIST LBRACKET ty RBRACKET
      { TyList $3 }
```