Checking Tail Recursion in PicoML

Umang Mathur
umathur3@illinois.edu

Chia-Hao Hsieh
chsieh17@illinois.edu

December 17, 2015

1 Overview

1.1 Recursion

The use of recursion dates back to the late 19^{th} century, when mathematicians Dedekind and Peano used induction to defined functions. The use of recursion played an important role in foundations of computer science, and was later referred to as 'primitive recursion' [1]

Use of recursion is not just exciting from the perspective of a Mathematician, but is also quite significant from the perspective of a developer. Allowing procedures to be recursive helps the programmer write more readable and intuitive/natural programs. A notable use of recursion is seen when dealing with inductive structures. Inductive definitions and inductive programs can be very naturally programmed as recursive functions. Besides, recursive functions, can be easier to debug, due to the same reason. Recursive programs, at times, tend to be more efficient than a naive program with loops and no recursive calls. Recursion, thus is a very handy tool for programmers.

1.2 Checking Tail Reclusion : Motivation

The convenience offered due to recursion, comes at a cost. Recursive programs are generally modelled by the use of stack frames. This means that recursive programs tend to consume extra space (stack) for every recursive call they make. Besides, the additional overhead of copying the variables and values to the new frame, also accounts for a non trivial overhead, which at times, is not desirable from the standpoint of efficiency.

However, the extra space consumed can be overcome when the recursive call is the last thing the function does. In this case, the contents of the stack can be replaced by the new frame, and there is no need to push an additional frame.

The idea behind tail call optimization is essentially the same. Informally, a recursive function is tail recursive when the recursive call is the last thing executed by the function. Thus, if the compiler can detect if a function is tail-recursive, it can convert the function to an equivalent while-loop, thus avoiding an additional call that consumes extra stack space by virtue of the new frame added.

1.3 Goal of the Project

In this project, we implement a tool **`TailRec`** that checks if a procedure is tail recursive or not. Specifically, we wish to analyze declarations written in PicoML. PicoML is a restricted form of OCaml, and supports simple expressions like `if then else`, `fun`, `letrec`. As part of the assignments in the course, we have built an interpreter for this language [2]. We aim to integrate the **`TailRec`** with the interpreter. That is, we would use the parsing and the type checking functionality written in older assignments. This would enable use to directly use the functionality for implementing **`TailRec`** , and would save some effort, as compared to the scenario where we had to re-invent the wheel.

2 Definitions

Before we describe our implementation, it would be useful to go through some notations and definitions. The purpose of the definitions is to give a nice characterization of the problem we wish to address.

Definition 2.1 (PicoML Expression)**.** A PicoML-style expression e is formally defined by the recursive grammar:

$$\begin{aligned} e := c \mid v \mid \ \odot e \mid e \oplus e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid e\ e \mid \texttt{fun}\ x\ \rightarrow\ e \\ \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{let rec}\ f\ x = x\ \texttt{in}\ e \mid \texttt{raise}\ e \\ \mid \texttt{try}\ e\ \texttt{with}\ \texttt{e\_int\_list} \end{aligned}$$

where, c is a constant, v is a variable, and `e_int_list` is inductively defined as:

$$\texttt{e\_int\_list} := \texttt{i} \rightarrow e \mid \texttt{i} \rightarrow e, \texttt{e\_int\_list}$$

where i is an integer

Definition 2.2 (PicoML Declaration)**.** Declarations in PicoML are declarative statements that assign an expression to an identifier. Alternately, then can also be plain expressions.

$$\texttt{dec} := e \mid \texttt{let}\ x = e \mid \texttt{let rec}\ f\ x = e$$

Definition 2.3 (Recursive Expression)**.** An expression e defined in PicoML is defined to be recursive with respect to an identifier f if there is a subexpression e' of e (that is not lambda lifted) and has the form $f\ e''$, where e'' is an expression, and no expression that contains e' redefines f

Let us take a look at a couple of examples to understand the above definition.

Example 2.1. Consider the PicoML declaration below:

$$\texttt{let rec}\ f\ x\ =\ \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ f\ (f\ x - 1);;$$

Note that, the body of the above declaration is recursive in f, based on the above definition.

Example 2.2. Consider the expression defined below:

$$\texttt{let}\ g\ = f\ \texttt{in}\ g\ 3;;$$

Note that, based on the definition above, the above expression is not recursive in f

Definition 2.4. Tail Recursive Expression A PicoML expression e is said to be tail-recursive with respect to an identifier f if it is recursive in f and one of the following holds:

- e is a constant c or a variable v
- e is of the form $\odot e'$, and e' is not recursive in f
- e is of the form $e' \oplus e''$, and none of e' and e'' is recursive in f
- e is of the form $e'e''$ and e'' is not recursive in f and e' is tail-recursive in f
- e is of the form $\texttt{if}\ e'\ \texttt{then}\ e''\ \texttt{else}\ e'''$ if e' is not recursive in f, and both e'' and e''' are tail recursive in f
- e is of the form $\texttt{fun}\ x\ \rightarrow\ e'$

- e is of the form `let` $x = e'$ `in` e'' and, either
  - x is not f (that is, f has the older binding in e''), e' is not recursive in f, and e'' is tail recursive in f, or
  - x is f (that is, the binding of f is updated in e''), and e' is tail recursive in f
- e is of the form `let rec` $g\ x = x$ `in` e', and, one of the following is true
  - g is f (that is, f is given a new binding in e''), or
  - g is not f, (here, f retains its binding in e''), and e'' is tail recursive in f
- e is of the form `try` e' `with` $i1 \to e_{i1} \mid i2 \to e_{i2} \mid \ \ldots \mid ik \to e_{ik}$, and
  - e' is not tail recursive in f, and
  - Each of e_{i1} through e_{ik} is tail recursive in f

Note that, as discussed in Example 2.2, the above characterization does not check for tail recursion if the expression is tail recursive by virtue of aliases. While this takes our characterization further from being a complete characterization, we do not compromise the soundness. That is to say, that if an expression is reported to tail recursive using our characterization, it cannot be a false alarm.

Definition 2.5 (Tail Recursive Declaration)**.** A PicoML declaration is said to be tail recursive if one of the following hold:

- It is of the form e
- It is of the form `let` $x = e$
- It is of the form `let rec` $f\ x = e$ and e is tail recursive in f

3 Implementation

For the purpose of the project, we check if a PicoML declaration is tail recursive or not, in the following two ways :

1. We check if the declaration satisfies Definition 2.5 given earlier
2. We transform the declaration into Continuation Passing style, and check for some conditions (covered later)

3.1 Checking Tail Recursion in PicoML

For the, purpose of checking tail recursion in **`TailRec`** we need to ensure that whenever an expression satisfies the conditions in Definition 2.4, we should report so, and report 'not tail recursive' otherwise. In order to ensure that this is the case, our implementation has to be faithful to the definition. Because of the inductive nature of the definition, it becomes lot easier to implement the algorithm.

Here is a brief snapshot of the algorithm:

```
let rec check_tail_rec_f f e =
    match e
    with ConstExp c -> true
    | VarExp v -> true
    | MonOpAppExp (mon_op, e1) -> not (check_rec_f f e1)
    | BinOpAppExp (bin_op, e1, e2) -> (not (check_rec_f f e1)) && (not (check_rec_f f e2))
    | AppExp(e1, e2) -> if (check_rec_f f e2) then false else check_tail_rec_f f e1
    | IfExp (e1, e2, e3) -> (not (check_rec_f f e1)) && (check_tail_rec_f f e2) && (
    check_tail_rec_f f e3)
    | FunExp (x, e1) -> true
    | LetInExp (x, e1, e2) -> if (x = f) then (not (check_rec_f f e1)) else ( (not (check_rec_f f
     e1)) && (check_tail_rec_f f e2))
    | LetRecInExp (g, x, e1, e2) ->
        if (g = f) then true
        else if (not (g=f) && (x=f)) then (check_tail_rec_f f e2)
        else ( (check_tail_rec_f f e2))
    | TryWithExp (e', n1opt, e1, nopt_e_lst) ->
        if (check_rec_f f e') then false
            else let lst = ((n1opt, e1)::nopt_e_lst) in (List.fold_right (fun (intop, h) -> fun t
     -> (check_tail_rec_f f h) && t) lst true)
    | _ -> false ;;
```

Listing 1: Tail recursion for PicoML expressions

3.2 Checking Tail Recursion in CPS expressions

Checking if a CPS transformed expression is tail recursive in f basically amounts to checking if an application of f is passed onto its original continuation (which happens to be a trivial check for our framework, because the original continuation is an artificially added dummy continuation, having a special signature).

The idea behind this is that if the continuation for an application of f is another function, it basically means that the application is not in the tail position, and there is some *work to be done* after the application.

Below is a snippet of the relevant part of the code:

```
let rec check_cps_tail_rec_f flist original_k x k e =
    match e
    with ConstCPS (k', c) -> cont_tail_recursive original_k k' flist
    | VarCPS (k', v) -> cont_tail_recursive original_k k' flist
    | MonOpAppCPS (k', mono_op, o1, exk) -> cont_tail_recursive original_k k' flist
    | BinOpAppCPS (k', bin_op, o1, o2, exk) -> cont_tail_recursive original_k k' flist
    | IfCPS (b, e1, e2) -> (check_cps_tail_rec_f flist original_k x k e1) && (
    check_cps_tail_rec_f flist original_k x k e2)
    | AppCPS (k', e1, e2, exk) ->
        if ( List.exists (fun x -> x = e1) flist)
        then (k'=original_k)
```

```
        else cont_tail_recursive original_k k' flist
    | FunCPS (kappa, x, k, ek, e) -> true
    | FixCPS (kappa, f, x, k, ek, e) -> true
and cont_tail_recursive original_k k flist =
    match k
    with ContVarCPS i -> true
    | External -> true
    | FnContCPS (x, e) ->
        check_cps_tail_rec_f flist original_k x k e
    | ExnMatch ek -> true ;;
```

Listing 2: Tail recursion for CPS transformed expressions

3.3 Code Structure

The tool **TailRec** has been developed in a series of iterations, and deliberate attempts have been made to keep the code concise and short. The total lines of code is about 1800, and the central parts of the code (where we implement the main algorithms), is about 200 lines (100 lines each for direct style and CPS style)

The code has been kept modular, to the best possible extent. The following is a brief description of the various files:

- `definitions.ml` ; Includes the various definitions for the types used in the tool. This file includes algorithms for unification, type-inference, transforming PicoML expressions in CPS etc., apart from basic utilities for printing various types. Most of the parts in this file have been derived from MP7
- `tailRecPicoMLlex.mll` : Includes the code for token generation for PicoML expressions, largely borrowed from ML4
- `tailRecPicoMLparse.mly` : Includes the code for parsing PicoML expressions, largely borrowed from MP5
- `checkTailRec.ml` : Contains the main algorithm for checking PicoML expressions and declarations for tail recursion
- `checkTailRecCPS.ml` : Contains the main algorithm for checking CPS expressions and declarations for tail recursion
- `tailRecPicoMLInt.ml` : Contains utilities for wrapping the main code for interactive use
- `tailRecPicoMLTest.ml` : Contains utilities for wrapping the main code for non-interactive testing

3.4 Comparison with Original Proposal

Below is the relevant part of the original proposal :

```
Title: Checking Tail Recursion in PicoML

Team:
1. Umang Mathur     umathur3@illinois.edu
2. Chia-hao Hsieh   chsieh17@illinois.edu

We intend to check if a program defined in PicoML, having the form
    let rec f = RHS_expression      -- (1)
is tail recursive or not.

* Given an expression e and identifier f, e is said to be "recursive in f",
if it has a sub-expression of the form AppExp (f, e1), where e1 is a well formed expression.

* Given an expression e and identifier f, e is said to be "tail-recursive in f",
if it is "recursive in f" and one of the following holds:

1. e = AppExp(e1, e2). and
    a. e2 is not "recursive in f"
```

```
    b. e1 = f
       or
       e1 is "tail-recursive in f"

2. e = IfExp (e1, e2, e3), and
    a. e1 is not "recursive in f"
    b. If e2 is either "tail-recursive in f" or not "recursive in f", and e3 is "tail-recursive in f"
       or
       If e3 is either "tail-recursive in f" or not "recursive in f", and e2 is "tail-recursive in f"

3. e = FunExp (x, e1), and
    a. e1 is "tail-recursive is f"

4. e = LetIn (x, e1, e2), and
    a. e1 is not "recursive is f"
    b. e2 is "tail-recursive in f"

5. e = LetRecIn (g, x, e1, e2), and
    a. e1 is not "recursive is f"
    b. e2 is "tail-recursive in f"

6. e = TryWithExp(e’, (i1, e1), (i2, e2), ..... (ik, ek)), and
    a. e’ is not "recursive in f"
    b. If each of e1, ..., ek is either "tail-recursive in f", or is  not "recursive in f"
    c. At least one of e1, ... ek, is "tail-recursive" in f.


* A program defined as (1) is "tail-recursive" if RHS_expresison is "tail-recursive in f"

Hence, for checking if any program of the form (1) is "tail-recursive" or not,
we will evaluate the parse tree (Abstract Syntax Tree) of the RHS_expression
and check if it is "recursive in f" and also "tail-recursive in f"

Breakup:

Umang Mathur:
    1. Implementation of cases 1., 2., 3.
    2. Testing for cases 4., 5., 6.

Chia-Hao Hsieh:
    1. Implementation of cases 4., 5., 6.
    2. Testing for cases 1., 2., 3.

Timeline:
    1. December 9, 2015 : Implementation
    2. December 11, 2015: Testing
    2. December 16, 2015: Report, and, Presentation

General Comments:
We believe the definition of a tail-recursive expresison may not be exhaustive,
and we might be missing some obvious cases.
In particular, we would like to ask if the following functions are tail recursive:

1. let rec f n =    if n = 0
                    then 1
                    else
                        if (f 0) = 1
                        then f (n-1)
                        else 2;;

  Is f tail-recursive ?
  The issue here is that the else expression of the outer "if"
  makes a mandatory call to f (in checking "(f 0) = 1").
  According to our definition above, the above function is not tail-recursive

2. let rec f n =    if n = 0
```

```
                    then 1
                    else f (n*(f(n-1)));;
  Is f tail-recursive ?
  The issue here is that the last operation is a call to f, so should it be called a tail call ?
  According to our definition above, the above function is not tail-recursive

3. let rec g n = if n = 0 then 1 else n * (g (n-1))
  let rec f n = g n;;
  Is f tail-recursive ?
  According to our definition above, the above function is indeed tail-recursive


4. let rec f x aux =
    let g = f
    in
    if x = 0 then aux else g (x-1) (x*aux);;

Is the above declaration tail recursive ?
It does not even fit in our definition of "recursive in f".
```

Clearly, we have improved upon the proposal in the following ways:

1. We have changed a lot of definitions, realizing that the earlier characterizations were unsound, or far from complete, or completely wrong. Specific differences have not been pointed out for the sake of conciseness

2. We have also implemented an algorithm for checking CPS expressions, thus enabling us to check for robustness of the tool.

4 Tests

Each line in the file `testing.txt` is a test case. To see testing results, run `./tailRecPicoMLTest` after doing a `make`

Test cases can be categorized as follows:

4.1 Smoke Tests

```
5;; (* for dec = (Anon e) *)
let f = 5;; (* for dec = Let (s, e) *)
let rec f x = 100;; (* smoke test for ConstExp *)
let rec f x = x;; (* smoke test for VarExp *)
let rec f x = hd x;; (* smoke test for MonOp *)
let rec f x = x + x;; (* smoke test for BinOp *)
let rec f x = f 5;; (* smoke test for AppExp *)
let rec f x = (fun x -> x);; (* smoke test for FunExp *)
```

Listing 3: Smoke Tests

4.2 Typical Example Cases

```
let rec f x = if (x=0) then 0 else f (x-1);; (* typical tail-recursive example *)
let rec f x = if (x=0) then 0 else x * (f (x-1));; (* typical not tail-recursive example *)
```

Listing 4: Typical Example Cases

4.3 More Test Cases

```
let rec f x = if (x=0) then 0 else f(f(f(f (x-1))));; (* a variavation of typical tail-recursive
    example *)
let rec f x = let f = 5 in f + f;; (* for LetIn: redefine f *)
let rec f x = if (x=0) then 0+0+0+0 else (f (x-1));; (* for BinOp *)
let rec f x = (fun y -> 0) x;; (* more cases for FunExp and AppExp *)
let rec f x = (fun x -> x * x) x ;;
let rec f x = (fun x -> x * (f x)) x ;;
let rec f x = let x_0=(x=0) in if x_0 then 0 else (f (x-1));; (* more cases for LetIn *)
let rec f x = let f_x_1=(f (x-1)) in if x = 0 then 0 else f_x_1;;
let rec f x = let g = (f (x-1)) in if (x=0) then 0 else g;;
let rec f x = let g = x-1 in if (x=0) then 0 else f g;;
let rec f x = let g = (f x) + (f x) in if (x=0) then 0 else (f (x-1));;
let rec f x = let g = (f x) + (f x) in if (x=0) then 0 else g;;
let rec f x = let rec g y = y * y in if (x=0) then (g 0) else (f (x-1));; (* more cases for
    LetRecIn *)
let rec f x = let rec f y = 0 in if (x=0) then (f 0) else (f (x-1));;
let rec f x = let rec g f = 0 in if (x=0) then (g 0) else (f (x-1));;
let rec f x = let rec g y = (f y) in if (x=0) then 0 else (f (x-1));;
let rec f x = let rec g y = (f y) + (f y) in if (x=0) then 0 else g x;;
let rec f x = let rec g y = (f y) in if (x=0) then 0 else (g (x-1));;
```

Listing 5: More Test Cases

4.4 Test Cases that aren't handled well

The following test cases, are known to be either failing, or pass without the correct reason (that is, they do not fit out characterization for the correct reason)

```
let rec f x = let f = f in if (x=0) then 0 else (f (x-1));; (* more cases for LetIn: aliases *)
let rec f x = let g = f in if (x=0) then 0 else g (x-1);;
let rec f x = let g = f in g x;;
let rec f x = let g = f in if (x=0) then 0 else (g (x-1));;
let rec f x = let g = f in if (x=0) then 0 else (x-1)*(g (x-1));; (* Failed with both styles
    since we can't handle aliases well *)
let rec f x = let g = f in let h = g in if (x=0) then 0 else (h (x-1))*(x-1);; (* Failed with
    both styles since we can't handle aliases well *)
```

Listing 6: Unhandled Test Cases

5 Listing

This sections gives a listing of the code written for the tool **TailRec**

5.1 checkTailRec.ml

```
open Definitions;;

let rec check_rec_f f e =
    (* check if there is AppExp(f, ...) in e *)
    match e
    with ConstExp c -> false
    | VarExp v -> false
    | MonOpAppExp (mon_op, e1) -> check_rec_f f e1
    | BinOpAppExp (bin_op, e1, e2) -> (check_rec_f f e1) || (check_rec_f f e2)
    | IfExp (e1, e2, e3) ->
        (check_rec_f f e1) || (check_rec_f f e2) || (check_rec_f f e3)
    | LetInExp (s, e1, e2) ->
        if (check_rec_f f e1)
            then true
            else
                (
                if (s=f) then false else ( (check_rec_f f e1) || (check_rec_f f e2) )
                )
    | FunExp (s, e1) -> false
    | AppExp (e1, e2) ->
        (
        match e1
        with VarExp v -> if (v=f) then true else false
        | _ -> (check_rec_f f e1) || (check_rec_f f e2)
        )
    | LetRecInExp (g, x, e1, e2) ->
        if (g=f)
            then false
            else (check_rec_f f e2)
    | RaiseExp e1 -> (check_rec_f f e1)
    | TryWithExp (e0, n1opt, e1, nopt_e_lst) ->
        (check_rec_f f e0) || (check_rec_f_lst f ((n1opt, e1) :: nopt_e_lst) )

and check_rec_f_lst f nopt_e_lst =
    match nopt_e_lst
    with [] -> true
    | ((nnopt, en)::rest) -> ((check_rec_f f en) || (check_rec_f_lst f rest));;

let rec check_tail_rec_f f e =
    match e
    with ConstExp c -> true (* it is not recursive, so it is tail-recrusive  *)
    | VarExp v -> true (* it is not recursive, so it is tail-recrusive  *)
    | MonOpAppExp (mon_op, e1) ->
        (* if f is not called in e1, then it is not recursive, and therefore tail-recrusive  *)
        not (check_rec_f f e1)
    | BinOpAppExp (bin_op, e1, e2) ->
        (* if f is not called in e1 or e2, then it is not recursive, and therefore tail-recrusive *)
        (not (check_rec_f f e1)) && (not (check_rec_f f e2))
    | AppExp(e1, e2) -> if (check_rec_f f e2)
        then (* if f is called in e2, that call is not a tail call. So it is not tail-recursive. *)
            false
        else
            check_tail_rec_f f e1
    | IfExp (e1, e2, e3) ->
            (* if f is called in e1, that call is not a tail call. So it is not tail-recursive. *)
            (not (check_rec_f f e1)) &&
            (check_tail_rec_f f e2) &&
            (check_tail_rec_f f e3)
```

```

    | FunExp (x, e1) ->
        (* Even if there is a cell to f in e1,
        it is still tail-recursive because that call is not available in the body of f. *)
        true

    | LetInExp (x, e1, e2) ->
        if (x = f)
            then
                (* if x is f, then we don't have to worry about whether there is f in e2.
                Because even there is, that f is not the original f itself.
                In this case, we only need to make sure f is not called in e1
                since f in e1 is not a tail call. *)
                (not (check_rec_f f e1))
            else
                (* if f is not called in e', then whethere it is tail-recursive is determined by
    e2. *)
                ( (not (check_rec_f f e1)) && (check_tail_rec_f f e2))
    | LetRecInExp (g, x, e1, e2) ->
        if (g = f)
            then
                (* if g is f, then we don't have to worry about whether there is f in e2.
                Because even there is, that f is not the original f itself.  *)
                true
        else if (not (g=f) && (x=f))
                then
                    (* if x is f, then all f's in e1 is just x, not the original f. *)
                    (check_tail_rec_f f e2)
        else
            (* g is just like another function, like the FunExp case. *)
            ( (check_tail_rec_f f e2))

    | TryWithExp (e', n1opt, e1, nopt_e_lst) ->
        (
        if (check_rec_f f e')
            then (* if f is called in e', that call is not a tail call. So it is not tail-
    recursive. *)
                false
            else let lst = ((n1opt, e1)::nopt_e_lst)
                in
                (* All e's in the list has to be tail-recursive in order to make TryWith to be
    tail-recursive. *)
                (List.fold_right (fun (intop, h) -> fun t -> (check_tail_rec_f f h) && t) lst
    true)
        )
    | _ -> false ;;


let check_tail_recursion_direct dec =
    match dec
    with (Anon e) -> true
    | Let (s, e) -> true
    | LetRec (f, x, e) ->
        check_tail_rec_f f e ;;
```

Listing 7: checktailRec.ml

5.2 checkTailRecCPS.ml

```
open Definitions

let rec convert_f_exp e name_of_f =
    (* Return all variable numbers that correspond to f. *)
    (
    match e
    with  ConstCPS (k, c) ->
        convert_f_cont k name_of_f
    | VarCPS (k, g) ->
            (
            match k
            with FnContCPS (number_of_f, e') ->
                let rec_list = convert_f_exp e' name_of_f
                in
                if (g = name_of_f)
                    (* Found a matched f. Add the number to retured list. *)
                    then (number_of_f :: rec_list)
                    else rec_list
            | _ -> []
            )
    | MonOpAppCPS (k, _, _, _) ->
        convert_f_cont k name_of_f
    | BinOpAppCPS (k , _, _, _, _) ->
        convert_f_cont k name_of_f
    | IfCPS (b, e1, e2) ->
        (convert_f_exp e1 name_of_f)@(convert_f_exp e2 name_of_f)
    | AppCPS (k, _, _, _) ->
        convert_f_cont k name_of_f
    | FunCPS (k, _, _, _, _) ->
        convert_f_cont k name_of_f
    | FixCPS (k, _, _, _, _, _) ->
        convert_f_cont k name_of_f
    )

and

convert_f_cont k name_of_f =
    (
    match k
    with FnContCPS (_, e') ->
        convert_f_exp e' name_of_f
    | _ -> []
    ) ;;


let rec check_cps_tail_rec_f flist original_k x k e =
    match e
    with ConstCPS (k', c) ->
        (* It's impossilbe to call f here, so go deeper to find if there are any f's. *)
        cont_tail_recursive original_k k' flist
    | VarCPS (k', v) ->
        (* Similarly, go deeper to find f. *)
        cont_tail_recursive original_k k' flist
    | MonOpAppCPS (k', mono_op, o1, exk) ->
        (* Similarly, go deeper to find f. *)
        cont_tail_recursive original_k k' flist
    | BinOpAppCPS (k', bin_op, o1, o2, exk) ->
        (* Similarly, go deeper to find f. *)
        cont_tail_recursive original_k k' flist
    | IfCPS (b, e1, e2) ->
        (* Similarly, go deeper to find f. *)
        (check_cps_tail_rec_f flist original_k x k e1) && (check_cps_tail_rec_f flist original_k
    x k e2)
    | AppCPS (k', e1, e2, exk) ->
        (
        (* if e1 matches one of variable numbers corresponding to f ... *)
```

```
        if ( List.exists (fun x -> x = e1) flist)
            then
                (
                (* And if the returned k is the original ...*)
                if (k'=original_k)
                    then true (* then this is a tail call. *)
                    else false
                )
            else
                (* Else just go deeper to find if there are any f's. *)
                cont_tail_recursive original_k k' flist
        )
    | FunCPS (kappa, x, k, ek, e) -> true
    | FixCPS (kappa, f, x, k, ek, e) -> true

and cont_tail_recursive original_k k flist =
    match k
    with ContVarCPS i -> true
    | External -> true
    | FnContCPS (x, e) ->
        check_cps_tail_rec_f flist original_k x k e
    | ExnMatch ek -> true ;;


let check_tail_recursion_cps dec =
    match dec
    with Anon e -> true
    | Let (x,e) -> true
    | LetRec (f,x,e) ->
        let (i,j) = (next_index(),next_index())
        in
            let ecps2 = cps_exp e (ContVarCPS i) (ExnContVarCPS j)
            in

            let flist = convert_f_exp ecps2 f
                in
                check_cps_tail_rec_f flist (ContVarCPS i) x (ContVarCPS i) ecps2 ;;
```
Listing 8: checlTailRecCPS.ml

5.3 definitions.ml

```
(* File: mp7common.ml *)

(* expressions for PicoML *)
type const = BoolConst of bool | IntConst of int | FloatConst of float
           | StringConst of string | NilConst | UnitConst

let string_of_const c =
    match c
    with IntConst n    -> if n < 0 then "~"^string_of_int(abs n) else string_of_int n
       | BoolConst b   -> if b then "true" else "false"
       | FloatConst f  -> string_of_float f
       | StringConst s -> "\"" ^ s ^ "\""
       | NilConst      -> "[]"
       | UnitConst     -> "()"

type mon_op = IntNegOp | HdOp | TlOp | FstOp | SndOp | PrintStringOp

let string_of_mon_op m =
    match m
    with IntNegOp   -> "~"
       | HdOp  -> "hd"
       | TlOp  -> "tl"
       | FstOp   -> "fst"
       | SndOp   -> "snd"
       | PrintStringOp  -> "print_string"

type bin_op = IntPlusOp | IntMinusOp | IntTimesOp | IntDivOp
           | FloatPlusOp | FloatMinusOp | FloatTimesOp | FloatDivOp
           | ConcatOp | ConsOp | CommaOp | EqOp | GreaterOp
           | ModOp | ExpoOp

let string_of_bin_op = function
     IntPlusOp  -> " + "
   | IntMinusOp -> " - "
   | IntTimesOp -> " * "
   | IntDivOp -> " / "
   | FloatPlusOp -> " +. "
   | FloatMinusOp -> " -. "
   | FloatTimesOp -> " *. "
   | FloatDivOp -> " /. "
   | ConcatOp -> " ^ "
   | ConsOp -> " :: "
   | CommaOp -> " , "
   | EqOp  -> " = "
   | GreaterOp -> " > "
   | ExpoOp -> "**"
   | ModOp   -> "mod"

type exp =  (* Exceptions will be added in later MPs *)
   | VarExp of string                    (* variables *)
   | ConstExp of const                   (* constants *)
   | MonOpAppExp of mon_op * exp         (* % e1 for % is a builtin monadic operator *)
   | BinOpAppExp of bin_op * exp * exp   (* e1 % e2 for % is a builtin binary operator *)
   | IfExp of exp * exp * exp            (* if e1 then e2 else e3 *)
   | AppExp of exp * exp                 (* e1 e2 *)
   | FunExp of string * exp              (* fun x -> e1 *)
   | LetInExp of string * exp * exp      (* let x = e1 in e2 *)
   | LetRecInExp of string * string * exp * exp (* let rec f x = e1 in e2 *)
   | RaiseExp of exp                     (* raise e *)
   | TryWithExp of (exp * int option * exp * (int option * exp) list)
                       (* try e with i -> e1 | j -> e1 | ... | k -> en *)

type dec =
     Anon of exp
   | Let of string * exp                 (* let x = exp *)
   | LetRec of string * string * exp     (* let rec f x = exp *)
```

```
let rec string_of_exp = function
   VarExp s -> s
 | ConstExp c ->  string_of_const c
 | IfExp(e1,e2,e3)->"if " ^ (string_of_exp e1) ^
                  " then " ^ (string_of_exp e2) ^
                  " else " ^ (string_of_exp e3)
 | MonOpAppExp (m,e) ->  (string_of_mon_op m) ^ " " ^ (paren_string_of_exp e)
 | BinOpAppExp (b,e1,e2) ->
   (match b with CommaOp -> ("(" ^ (paren_string_of_exp e1) ^ (string_of_bin_op b) ^
                              (paren_string_of_exp e2) ^ ")")
    | _ -> ((paren_string_of_exp e1) ^ " " ^ (string_of_bin_op b)
           ^ " " ^ (paren_string_of_exp e2)))
 | AppExp(e1,e2) -> (non_app_paren_string_of_exp e1) ^ " " ^ (paren_string_of_exp e2)
 | FunExp (x,e) ->  ("fun " ^ x ^ " -> " ^ (string_of_exp e))
 | LetInExp (x,e1,e2) -> ("let "^x^" = "^ (string_of_exp e1) ^ " in " ^ (string_of_exp e2))
 | LetRecInExp (f,x,e1,e2) ->
    ("let rec "^f^" "^x^" = "^(string_of_exp e1) ^ " in " ^ (string_of_exp e2))
 | RaiseExp e -> "raise " ^ (string_of_exp e)
 | TryWithExp (e,intopt1,exp1,match_list) ->
    "try " ^ (paren_string_of_exp e) ^  " with " ^
     (string_of_exc_match (intopt1,exp1)) ^
     (List.fold_left (fun s m -> (s^" | " ^ (string_of_exc_match m))) "" match_list)

and paren_string_of_exp e =
    match e with VarExp _ | ConstExp _ -> string_of_exp e
    | _ -> "(" ^ string_of_exp e ^ ")"

and non_app_paren_string_of_exp e =
    match e with AppExp (_,_) -> string_of_exp e
    | _ -> paren_string_of_exp e


and string_of_exc_match (int_opt, e) =
    (match int_opt with None -> "_" | Some n -> string_of_int n) ^
    " -> " ^
    (string_of_exp e)

let string_of_dec = function
 | Anon e -> ("let _ = "^ (string_of_exp e))
 | Let (s, e) ->  ("let "^ s ^" = " ^ (string_of_exp e))
 | LetRec (fname,argname,fn) ->
    ("let rec " ^ fname ^ " " ^ argname ^ " = " ^ (string_of_exp fn))

let print_exp exp = print_string (string_of_exp exp)
let print_dec dec = print_string (string_of_dec dec)

(* Util functions *)
let rec drop y = function
   []    -> []
 | x::xs -> if x=y then drop y xs else x::drop y xs

let rec delete_duplicates = function
   []    -> []
 | x::xs -> x::delete_duplicates (drop x xs)

(********************** Type System Infrastructure **********************)

type typeVar = int

let rec expand n (list,len) =
    let q = n / 26 in
        if q = 0 then (n :: list, len + 1)
        else expand q (((n mod 26)::list), len + 1);;

let string_of_typeVar n =
   let (num_list,len) =
       match (expand n ([],0))
```

```
       with ([],l) -> ([],l) (* can't actually happen *)
          | ([s],l) -> ([s],l)
          | (x::xs,l) -> ((x - 1) :: xs, l)
   in
   let s = (Bytes.create len)
   in
   let _ =
    List.fold_left
    (fun n c -> (Bytes.set s n c; n + 1))
    0
    (List.map (fun x -> Char.chr(x + 97)) num_list)  (* Char.code 'a' = 97 *)
   in "'"^s;;

type monoTy = TyVar of typeVar | TyConst of (string * monoTy list)

let rec string_of_monoTy t =
  let rec string_of_tylist = function
     []      -> ""
   | t'::[] -> string_of_monoTy t'
   | t'::ts -> string_of_monoTy t'^ ","^ string_of_tylist ts
  in
  let string_of_subty s =
  match s with
     TyConst ("*", _) | TyConst ("->", _) -> ("("^ string_of_monoTy s^ ")")
   | _ ->  string_of_monoTy s
  in
    match t with
       TyVar n         -> (string_of_typeVar n)
     |TyConst (name, []) -> name
     |TyConst (name, [ty]) -> (string_of_subty ty^ " "^ name)
     |TyConst ("*", [ty1; ty2]) -> (string_of_subty ty1^ " * "^ string_of_monoTy ty2)
     |TyConst ("->", [ty1; ty2]) -> (string_of_subty ty1^ " -> "^ string_of_monoTy ty2)
     |TyConst (name, tys) -> ("("^ string_of_tylist tys^ ") "^ name)

let rec accummulate_freeVarsMonoTy fvs ty =
    match ty
    with TyVar n -> n::fvs
       | TyConst (c, tyl) -> List.fold_left accummulate_freeVarsMonoTy fvs tyl

let freeVarsMonoTy ty = delete_duplicates (accummulate_freeVarsMonoTy [] ty)

(*fresh type variable*)
let (fresh, reset) =
   let nxt = ref 0 in
   let f () = (nxt := !nxt + 1; TyVar(!nxt)) in
   let r () = nxt := 0 in
    (f, r)

let bool_ty = TyConst("bool",[])
let int_ty = TyConst ("int", [])
let float_ty = TyConst ("float",[])
let string_ty = TyConst ("string",[])
let unit_ty = TyConst("unit", [])
let mk_pair_ty ty1 ty2 = TyConst("*",[ty1;ty2])
let mk_fun_ty ty1 ty2 = TyConst("->",[ty1;ty2])
let mk_list_ty ty = TyConst("list",[ty])

type polyTy = typeVar list * monoTy  (* the list is for quantified variables *)

let string_of_polyTy (bndVars, t) = match bndVars with [] -> string_of_monoTy t
    | _ ->  (List.fold_left
             (fun s v -> s ^ " " ^ string_of_typeVar v)
             "Forall"
             bndVars)
             ^ ". " ^ string_of_monoTy t

let freeVarsPolyTy ((tvs, ty):polyTy) = delete_duplicates(
    List.filter (fun x -> not(List.mem x tvs)) (freeVarsMonoTy ty))
```

```
let polyTy_of_monoTy mty = (([],mty):polyTy)

let int_op_ty = polyTy_of_monoTy(mk_fun_ty int_ty (mk_fun_ty int_ty int_ty))
let float_op_ty =
    polyTy_of_monoTy(mk_fun_ty float_ty (mk_fun_ty float_ty float_ty))
let string_op_ty =
    polyTy_of_monoTy(mk_fun_ty string_ty (mk_fun_ty string_ty string_ty))

(* fixed signatures *)
let const_signature const = match const with
   BoolConst b -> (([], bool_ty):polyTy)
 | IntConst n -> ([], int_ty)
 | FloatConst f -> ([], float_ty)
 | StringConst s -> ([], string_ty)
 | NilConst -> ([0],mk_list_ty (TyVar 0))
 | UnitConst -> ([], unit_ty)

let binop_signature binop = match binop with
     IntPlusOp   -> int_op_ty
   | IntMinusOp   -> int_op_ty
   | IntTimesOp   -> int_op_ty
   | IntDivOp   -> int_op_ty
   | ModOp      -> int_op_ty
   | ExpoOp     -> float_op_ty
   | FloatPlusOp   -> float_op_ty
   | FloatMinusOp   -> float_op_ty
   | FloatTimesOp   -> float_op_ty
   | FloatDivOp   -> float_op_ty
   | ConcatOp -> string_op_ty
   | ConsOp ->
       let alpha = TyVar 0
       in ([0],
              mk_fun_ty alpha (mk_fun_ty (mk_list_ty alpha) (mk_list_ty alpha)))
   | CommaOp ->
       let alpha = TyVar 0 in
       let beta = TyVar 1 in
           ([0;1],
            mk_fun_ty alpha (mk_fun_ty beta (mk_pair_ty alpha beta)))
(*   | EqOp -> ([],mk_fun_ty int_ty (mk_fun_ty int_ty bool_ty)) *)
   | EqOp ->
     let alpha = TyVar 0 in ([0],mk_fun_ty alpha (mk_fun_ty alpha bool_ty))
   | GreaterOp ->
     let alpha = TyVar 0 in ([0],mk_fun_ty alpha (mk_fun_ty alpha bool_ty))

let monop_signature monop = match monop with
    | HdOp -> let alpha = TyVar 0 in([0], mk_fun_ty (mk_list_ty alpha) alpha)
    | TlOp -> let alpha = TyVar 0 in
                  ([0], mk_fun_ty (mk_list_ty alpha) (mk_list_ty alpha))
    | PrintStringOp -> ([], mk_fun_ty string_ty unit_ty)
    | IntNegOp -> ([], mk_fun_ty int_ty int_ty)
    | FstOp -> let t1,t2 = TyVar 0,TyVar 1
             in ([0;1],mk_fun_ty (mk_pair_ty t1 t2) t1)
    | SndOp -> let t1,t2 = TyVar 0,TyVar 1
             in ([0;1],mk_fun_ty (mk_pair_ty t1 t2) t2)

(* environments *)
type 'a env = (string * 'a) list

let freeVarsEnv l = delete_duplicates (
    List.fold_right (fun (_,pty) fvs -> freeVarsPolyTy pty @ fvs) l [])

let string_of_env string_of_entry gamma =
  let rec string_of_env_aux gamma =
    match gamma with
       []        -> ""
     | (x,y)::xs -> x^ " : "^ string_of_entry y^
                    match xs with [] -> "" | _  -> ", "^
```

```
                                                    string_of_env_aux xs
  in
    "{"^ string_of_env_aux gamma^ "}"

let string_of_type_env gamma = string_of_env string_of_polyTy gamma

(*environment operations*)
let rec lookup mapping x =
  match mapping with
     []        -> None
   | (y,z)::ys -> if x = y then Some z else lookup ys x

type type_env = polyTy env

let make_env x y = ([(x,y)]:'a env)
let lookup_env (gamma:'a env) x = lookup gamma x
let sum_env (delta:'a env) (gamma:'a env) = ((delta@gamma):'a env)
let ins_env (gamma:'a env) x y = sum_env (make_env x y) gamma

(****************** Unification, Adapted from MP5 ***********************)

type substitution = (typeVar * monoTy) list

let subst_fun (s:substitution) n = (try List.assoc n s with _ -> TyVar n)

(*unification algorithm*)
(* Problem 1 *)
let rec contains n ty =
  match ty with
    TyVar m -> n=m
  | TyConst(st, typelst) ->
     List.fold_left (fun xl x -> if xl then xl else contains n x) false typelst;;

(* Problem 2 *)
let rec substitute ie ty =
  let n,sub = ie
  in match ty with
       TyVar m -> if n=m then sub else ty
     | TyConst(st, typelist) -> TyConst(st, List.map (fun t -> substitute ie t) typelist);;

let polyTySubstitute s (pty:polyTy) =
    match s with  (n,residue) ->
    (match pty with (bound_vars, ty) ->
           if List.mem n bound_vars then pty
           else ((bound_vars, substitute s ty):polyTy))


(* Problem 3 *)
let rec monoTy_lift_subst (s:substitution) ty =
  match ty with
    TyVar m -> subst_fun s m
  | TyConst(st, typelst) ->  TyConst(st, List.map (fun t -> monoTy_lift_subst s t) typelst);;

(* Problem 4 *)
let rec unify eqlst : substitution option =
  let rec addNewEqs lst1 lst2 acc =
    match lst1,lst2 with
      [],[] -> Some acc
    | t::tl, t'::tl' -> addNewEqs tl tl' ((t,t')::acc)
    | _ -> None
  in
  match eqlst with
    [] -> Some([])
    (* Delete *)
  | (s,t)::eqs when s=t -> unify eqs
    (* Eliminate *)
  | (TyVar(n),t)::eqs when not(contains n t)->
      let eqs' = List.map (fun (t1,t2) -> (substitute (n,t) t1 , substitute (n,t) t2)) eqs
```

```
      in (match unify eqs' with
           None -> None
         | Some(phi) -> Some((n, monoTy_lift_subst phi t):: phi))
    (* Orient *)
  | (TyConst(str, tl), TyVar(m))::eqs -> unify ((TyVar(m), TyConst(str, tl))::eqs)
    (* Decompose *)
  | (TyConst(str, tl), TyConst(str', tl'))::eqs when str=str' ->
      (match (addNewEqs tl tl' eqs) with
        None -> None
      | Some l -> unify l)
    (* Other *)
  | _ -> None
;;




(***************** Support Infrastructure for  MP6 **********************)

(*judgment*)
type judgment =
   ExpJudgment of type_env * exp * monoTy
 | DecJudgment of type_env * dec * type_env

let string_of_judgment judgment =
  match judgment with ExpJudgment(gamma, exp, monoTy) ->
        string_of_type_env gamma ^ " |= "^ string_of_exp exp ^
         " : " ^ string_of_monoTy monoTy
  | DecJudgment (gamma, dec, delta) ->
        string_of_type_env gamma ^ " |= "^ string_of_dec dec ^
         " : " ^ string_of_type_env delta

type proof = Proof of proof list * judgment

(*proof printing*)
let string_of_proof p =
  let depth_max = 10 in
  let rec string_of_struts = function
     []    -> ""
   | x::[] -> (if x then "|-" else "|-")  (* ??? *)
   | x::xs -> (if x then "  " else "| ")^ string_of_struts xs
  in let rec string_of_proof_aux (Proof(ant,conc)) depth lst =
    "\n"^ "  "^ string_of_struts lst^
    (if (depth > 0) then "-" else "")^
    let assum = ant in
      string_of_judgment conc ^
      if depth <= depth_max
         then string_of_assum depth lst assum
      else ""
  and string_of_assum depth lst assum =
    match assum with
       []     -> ""
     | p'::ps -> string_of_proof_aux p' (depth + 1) (lst@[ps=[]])^
                 string_of_assum depth lst ps
  in
    string_of_proof_aux p 0 []^ "\n\n"

let rec monoTy_rename_tyvars s mty =
    match mty with
      TyVar n -> (match lookup s n with Some m -> TyVar m | _ -> mty)
    | TyConst(c, tys) -> TyConst(c, List.map (monoTy_rename_tyvars s) tys)

let subst_compose (s2:substitution) (s1:substitution) : substitution =
    (List.filter (fun (tv,_) -> not(List.mem_assoc tv s1)) s2) @
    (List.map (fun (tv,residue) -> (tv, monoTy_lift_subst s2 residue)) s1)

let gen (env:type_env) ty =
    let env_fvs = freeVarsEnv env in
```

```
    ((List.filter (fun v -> not(List.mem v env_fvs)) (freeVarsMonoTy ty), ty):polyTy)

let freshInstance ((tvs, ty):polyTy) =
    let fresh_subst = List.fold_right (fun tv s -> ((tv,fresh())::s)) tvs [] in
    monoTy_lift_subst fresh_subst ty

let first_not_in n l =
    let rec first m n l =
        if n > 0 then
         if List.mem m l then first (m+1) n l else m :: (first (m+1) (n - 1) l)
        else []
    in first 0 n l

let alpha_conv ftvs (pty:polyTy) =
    match pty with (btvs, ty) ->
    (let fresh_bvars =
         first_not_in (List.length btvs) (ftvs @ (freeVarsPolyTy pty))
     in (fresh_bvars,
         monoTy_lift_subst (List.combine btvs (List.map (fun v -> TyVar v) fresh_bvars))
         ty))

let polyTy_lift_subst s pty =
  let rec fvsfun x r = match x with
    | TyVar n -> n :: r
    | TyConst (_, l) -> List.fold_right fvsfun l r
  in
  let fvs = List.fold_right fvsfun (snd(List.split s)) [] in
    let (nbvs, nty) = alpha_conv fvs pty in
    ((nbvs, monoTy_lift_subst s nty):polyTy)

let rec mk_bty_renaming n bty =
    match bty with [] -> ([],[])
    | (x::xs) -> (match mk_bty_renaming (n-1) xs
                  with (s,l) -> (((x,n) :: s), n :: l))

let polyTy_rename_tyvars s (bty, mty) =
    let (renaming,new_bty) = mk_bty_renaming (~-7) bty in
    (new_bty, monoTy_rename_tyvars s (monoTy_rename_tyvars renaming mty))

let env_rename_tyvars s (env: 'a env) =
    ((List.map
      (fun (x,polyTy) -> (x,polyTy_rename_tyvars s polyTy)) env): 'a env)

let env_lift_subst s (env:'a env) =
    ((List.map (fun (x,polyTy) -> (x,polyTy_lift_subst s polyTy)) env):'a env)


(*constraint list*)
type consList = (monoTy * monoTy) list


(*applying a substitution to a proof*)

(*applying a substitution to a proof*)
let rec proof_lift_subst f = function
    Proof(assum, ExpJudgment(gamma, exp, monoTy)) ->
    Proof(List.map (proof_lift_subst f) assum,
          ExpJudgment(env_lift_subst f gamma, exp, monoTy_lift_subst f monoTy))
 | Proof(assum, DecJudgment(gamma, dec, delta)) ->
    Proof(List.map (proof_lift_subst f) assum,
          DecJudgment(env_lift_subst f gamma, dec, env_lift_subst f delta))

let rec proof_rename_tyvars f = function
    Proof(assum, ExpJudgment(gamma, exp, monoTy)) ->
    Proof(List.map (proof_rename_tyvars f) assum,
          ExpJudgment(env_rename_tyvars f gamma, exp,
                      monoTy_rename_tyvars f monoTy))
 | Proof(assum, DecJudgment(gamma, dec, delta)) ->
```

```
   Proof(List.map (proof_rename_tyvars f) assum,
         DecJudgment(env_rename_tyvars f gamma, dec,
                     env_rename_tyvars f delta))

let get_ty = function
   None       -> raise(Failure "None")
 | Some(ty,p) -> ty

let get_proof = function
   None       -> raise(Failure "None")
 | Some(ty,p) -> p

let infer_exp gather_exp (gamma:type_env) (exp:exp) =
  let ty = fresh() in
  let result =
    match gather_exp gamma exp ty with
       None          -> None
     | Some(proof,sigma) -> match ty with
          | TyVar n -> Some (subst_fun sigma n, proof_lift_subst sigma proof)
          | _       -> None
  in let _ = reset() in
  result;;

let infer_dec gather_dec (gamma:type_env) (dec:dec) =
  let result =
    match gather_dec gamma dec with
       None -> None
     | Some(proof,sigma) -> Some (proof_lift_subst sigma proof)
  in let _ = reset() in
  result;;

let string_of_constraints c =
  let rec aux c =
     match c with
     | [] -> ""
     | [(s,t)] ->  (string_of_monoTy s^ " --> "^ string_of_monoTy t)
     | (s,t)::c' -> (string_of_monoTy s^ " --> "^ string_of_monoTy t^
         "; "^ aux c')
  in ("["^ aux c^ "]\n")


let string_of_substitution s =
  let rec aux s =
     match s with
     | [] -> ""
     | [(i,t)] -> ((string_of_typeVar i)  ^ " --> " ^ string_of_monoTy t)
     | (i,t)::s' -> (((string_of_typeVar i)  ^ " --> ")^
                    string_of_monoTy t^ "; "^ aux s')
  in ("["^ aux s^ "]\n")


let niceInfer_exp gather_exp (gamma:type_env) exp =
  let ty = fresh()
  in
  let result =
    match gather_exp gamma exp ty with
     None ->
      (print_string("Failure: No type for expression: "^
       string_of_exp exp^ "\n"^
       "in the environment: "^
       string_of_env string_of_polyTy gamma^ "\n");
       raise (Failure ""))
   | Some (p,s) ->
   (string_of_proof p^
  (*
   "Constraints: "^
   string_of_constraints c ^
   "Unifying..."^
```

```
   match unify c with
     None -> ("Failure: No solution for these constraints!\n"^
              raise (Failure ""))
    | Some s ->
  *)
   ("Unifying substitution: "^
    string_of_substitution s^
    "Substituting...\n"^
    let new_p = proof_lift_subst s p in
    string_of_proof new_p)) in
  let _ = reset() in
  result;;

let niceInfer_dec
    (gather_dec:(type_env -> dec -> (proof * type_env * substitution) option))
    (gamma:type_env) dec =
  let result =
    match gather_dec gamma dec with
     None ->
      (print_string("Failure: No type for declaraion: "^
       string_of_dec dec^ "\n"^
       "in the environment: "^
       string_of_env string_of_polyTy gamma^ "\n");
       raise (Failure ""))
   | Some (p,d,s) ->
   (string_of_proof p^
   ("Unifying substitution: "^
    string_of_substitution s^
    "Substituting...\n"^
    let new_p = proof_lift_subst s p in
    string_of_proof new_p)) in
  let _ = reset() in
  result;;

(* Collect all the TyVar indices in a proof *)

let rec collectTypeVars ty lst =
  match ty with
    TyVar m -> m::lst
  | TyConst(st, typelst) -> List.fold_left (fun xl x -> collectTypeVars x xl) lst typelst

let rec collectFreeTypeVars bty ty lst =
  match ty with
    TyVar m -> if List.mem m bty then lst else m::lst
  | TyConst(st, typelst) ->
    List.fold_left (fun xl x -> collectFreeTypeVars bty x xl) lst typelst

let collectPolyTyVars (bty,mty) lst = collectFreeTypeVars bty mty lst

let collectEnvVars (gamma:type_env) lst =
    List.fold_left (fun tys (_,pty)-> collectPolyTyVars pty tys) lst gamma

let collectJdgVars jdg lst =
    match jdg with ExpJudgment(gamma, exp, monoTy) ->
        collectEnvVars gamma (collectTypeVars monoTy lst)
    | DecJudgment(gamma, dec, delta) ->
        collectEnvVars gamma (collectEnvVars delta lst)

let rec collectProofVars prf lst =
  match prf with Proof (assum, jdg)
   -> collectAssumVars assum (collectJdgVars jdg lst)
and collectAssumVars assum lst =
  match assum with
    []     -> lst
  | p::ps -> collectAssumVars ps (collectProofVars p lst)

let canonicalize_proof prf_opt =
    match prf_opt with None -> None
```

```
    | Some(ty, prf) ->
  let (varlst,_) =
    List.fold_right (fun x (xl,idx) -> ((x,idx)::xl), idx+1)
      (delete_duplicates (collectProofVars prf (collectTypeVars ty [])))
      ([],1)
  in Some(monoTy_rename_tyvars varlst ty, proof_rename_tyvars varlst prf)

let canon = canonicalize_proof

let canon_dec prf_opt =
    match prf_opt with None -> None
    | Some prf ->
  let (varlst,_) =
    List.fold_right (fun x (xl,idx) -> ((x, idx)::xl), idx+1)
      (delete_duplicates (collectProofVars prf []))
      ([],1)
  in Some(proof_rename_tyvars varlst prf)

(******************** MP6's inferencer ********************)

let rec gather_exp_ty_substitution gamma exp tau =
    let judgment = ExpJudgment(gamma, exp, tau) in
    match exp
    with ConstExp c ->
         let tau' = const_signature c in
         (match unify [(tau, freshInstance tau')]
          with None       -> None
             | Some sigma -> Some(Proof([],judgment), sigma))
    | VarExp x ->
      (match lookup_env gamma x with None -> None
       | Some gamma_x ->
         (match unify [(tau, freshInstance gamma_x)]
          with None       -> None
             | Some sigma -> Some(Proof([],judgment), sigma)))
    | BinOpAppExp (binop, e1,e2) ->
      let tau' = binop_signature binop in
      let tau1 = fresh() in
      let tau2 = fresh() in
      (match gather_exp_ty_substitution gamma e1 tau1
       with None -> None
       | Some(pf1, sigma1) ->
         (match gather_exp_ty_substitution (env_lift_subst sigma1 gamma) e2 tau2
          with None -> None
          | Some (pf2, sigma2) ->
            let sigma21 = subst_compose sigma2 sigma1 in
            (match unify[(monoTy_lift_subst sigma21
                           (mk_fun_ty tau1 (mk_fun_ty tau2 tau)),
                         freshInstance tau')]
             with None -> None
             | Some sigma3 ->
               Some(Proof([pf1;pf2], judgment),subst_compose sigma3 sigma21))))
    | MonOpAppExp (monop, e1) ->
      let tau' = monop_signature monop in
      let tau1 = fresh() in
      (match gather_exp_ty_substitution gamma e1 tau1
       with None -> None
       | Some(pf, sigma) ->
         (match unify[(monoTy_lift_subst sigma (mk_fun_ty tau1 tau),
                       freshInstance tau')]
          with None -> None
          | Some subst ->
            Some(Proof([pf], judgment),
                 subst_compose subst sigma)))
    | IfExp(e1,e2,e3) ->
      (match gather_exp_ty_substitution gamma e1 bool_ty
       with None -> None
       | Some(pf1, sigma1) ->
         (match gather_exp_ty_substitution
```

```
                (env_lift_subst sigma1 gamma) e2 (monoTy_lift_subst sigma1 tau)
          with None -> None
          | Some (pf2, sigma2) ->
            let sigma21 = subst_compose sigma2 sigma1 in
            (match gather_exp_ty_substitution
                   (env_lift_subst sigma21 gamma) e3
                   (monoTy_lift_subst sigma21 tau)
             with  None -> None
             | Some(pf3, sigma3) ->
               Some(Proof([pf1;pf2;pf3], judgment), subst_compose sigma3 sigma21))))
    | FunExp(x,e) ->
      let tau1 = fresh() in
      let tau2 = fresh() in
      (match gather_exp_ty_substitution
             (ins_env gamma x (polyTy_of_monoTy tau1)) e tau2
       with None -> None
       | Some (pf, sigma) ->
         (match unify [(monoTy_lift_subst sigma tau,
                        monoTy_lift_subst sigma (mk_fun_ty tau1 tau2))]
          with None -> None
          | Some sigma1 ->
            Some(Proof([pf],judgment), subst_compose sigma1 sigma)))
    | AppExp(e1,e2) ->
      let tau1 = fresh() in
      (match gather_exp_ty_substitution gamma e1 (mk_fun_ty tau1 tau)
       with None -> None
       | Some(pf1, sigma1) ->
         (match gather_exp_ty_substitution (env_lift_subst sigma1 gamma) e2
                                           (monoTy_lift_subst sigma1 tau1)
          with None -> None
          | Some (pf2, sigma2) ->
            Some(Proof([pf1;pf2], judgment), subst_compose sigma2 sigma1)))
    | RaiseExp e ->
      (match gather_exp_ty_substitution gamma e int_ty
       with None -> None
       | Some(pf, sigma) -> Some(Proof([pf],judgment), sigma))
    | LetInExp(x,e1,e2)  ->
       let tau1 = fresh() in
       (match gather_exp_ty_substitution gamma e1 tau1
  with None -> None
     | Some(pf1, sigma1) ->
        let delta_env = make_env x (gen (env_lift_subst sigma1 gamma)
                (monoTy_lift_subst sigma1 tau1)) in
        (match gather_exp_ty_substitution
           (sum_env delta_env (env_lift_subst sigma1 gamma)) e2
                       (monoTy_lift_subst sigma1 tau)
         with None -> None
      | Some (pf2,sigma2) ->
         let sigma21 = subst_compose sigma2 sigma1 in
         Some(Proof([pf1;pf2], judgment), sigma21)))
    | LetRecInExp(f,x,e1,e2) ->
       let tau1  = fresh() in
       let tau2 = fresh() in
       let tau1_to_tau2 = mk_fun_ty tau1 tau2 in
       (match gather_exp_ty_substitution
    (ins_env (ins_env gamma f (polyTy_of_monoTy tau1_to_tau2))
        x (polyTy_of_monoTy tau1))
    e1 tau2
  with None -> None
     | Some(pf1, sigma1) ->
              let sigma1_gamma = env_lift_subst sigma1 gamma in
        let sigma1_tau1_to_tau2 = monoTy_lift_subst sigma1 tau1_to_tau2 in
        (match gather_exp_ty_substitution
                     (ins_env sigma1_gamma f (gen sigma1_gamma sigma1_tau1_to_tau2))
         e2 (monoTy_lift_subst sigma1 tau)
         with None -> None
      | Some(pf2,sigma2) ->
         let sigma21 = subst_compose sigma2 sigma1 in
```

```
         Some(Proof([pf1;pf2], judgment),sigma21)))
    | TryWithExp (e,intopt1,e1, match_list) ->
      (match (gather_exp_ty_substitution gamma e tau)
       with None -> None
       | Some (pf, sigma) ->
         (match
           List.fold_left
           (fun part_result -> fun (intopti, ei) ->
            (match part_result with None -> None
             | Some (rev_pflist, comp_sigmas) ->
               (match gather_exp_ty_substitution
                      (env_lift_subst comp_sigmas gamma) ei
                      (monoTy_lift_subst comp_sigmas tau)
                with None -> None
                | Some (pfi, sigmai) ->
                  Some (pfi :: rev_pflist, subst_compose sigmai comp_sigmas))))
           (Some([pf], sigma))
           ((intopt1,e1):: match_list)
           with None -> None
           | Some (rev_pflist, comp_subst) ->
             Some(Proof(List.rev rev_pflist, judgment), comp_subst)))


let rec gather_dec_ty_substitution gamma dec =
    match dec with
    | Anon e ->
      let tau = fresh() in
      (match gather_exp_ty_substitution gamma e tau
  with None -> None
     | Some(pf, sigma) ->
            Some(Proof([pf],DecJudgment (gamma, dec, [])), sigma))
    | Let(x,e) ->
       let tau = fresh() in
       (match gather_exp_ty_substitution gamma e tau
  with None -> None
     | Some(pf, sigma) ->
        let delta_env = make_env x (gen (env_lift_subst sigma gamma)
                (monoTy_lift_subst sigma tau)) in
            Some(Proof([pf],DecJudgment (gamma, dec, delta_env)),sigma))
    | LetRec(f,x,e) ->
       let tau1  = fresh() in
       let tau2 = fresh() in
       let tau1_to_tau2 = mk_fun_ty tau1 tau2 in
       (match gather_exp_ty_substitution
    (ins_env (ins_env gamma f (polyTy_of_monoTy tau1_to_tau2))
        x (polyTy_of_monoTy tau1))
     e tau2
  with None -> None
     | Some(pf, sigma) ->
             let sigma_gamma = env_lift_subst sigma gamma in
        let sigma_tau1_to_tau2 = monoTy_lift_subst sigma tau1_to_tau2 in
             let delta_env =
                (ins_env sigma_gamma f (gen sigma_gamma sigma_tau1_to_tau2))
             in
        Some(Proof([pf],DecJudgment (gamma, dec, delta_env)),sigma))

(***************** Adapted Infrstructure for MP5, CPS *********************)
type cps_cont =
   ContVarCPS of int                    (* _ki *)
 | External
 | FnContCPS of string * exp_cps        (* FN x -> exp_cps *)
 | ExnMatch of exn_cont                 (* i1 |-> ec1; ... in |-> ecn *)

and exn_cont =
   ExnContVarCPS of int
 | EmptyExnContCPS
 | UpdateExnContCPS of (int option * exp_cps) list * exn_cont

```

```
and exp_cps =
   VarCPS of cps_cont * string
 | ConstCPS of cps_cont * const
 | MonOpAppCPS of cps_cont * mon_op * string * exn_cont
 | BinOpAppCPS of cps_cont * bin_op * string * string * exn_cont
 | IfCPS of string * exp_cps * exp_cps
 | AppCPS of cps_cont * string * string * exn_cont
 | FunCPS of cps_cont * string * int * int * exp_cps
 | FixCPS of cps_cont * string * string * int * int * exp_cps

let (freshIntName , resetIntNameInt) =
    let intNameInt = ref 0 in
    let n() = (let x = !intNameInt in intNameInt := x + 1; string_of_int x) in
    let r() = intNameInt := 0 in
    (n,r)

let (next_index, reset_index) =
    let count = ref 0 in
    let n() = (let x = !count in count := x + 1; x) in
    let r() = count := 0 in
    (n,r)

let rec cps_exp e k ke =
   match e with
(*[[x]]k,ke = k x*)
     VarExp x -> (VarCPS (k, x))
(*[[c]]k,ke = k x*)
   | ConstExp c -> (ConstCPS (k, c))
(*[[~ e]]k,ke = [[e]](FN r ke -> k (~ r)), ke, *)
   | MonOpAppExp (m, e) ->
      let r = freshIntName() in
      cps_exp e ((*ClosedCPSCont*)(FnContCPS (r, MonOpAppCPS (k, m, r, ke)))) ke
(*[[(e1 + e2)]]k,ke = [[e1]](FN r -> [[e2]](FN s -> k (r + s)),ke),ke*)
   | BinOpAppExp (b, e1, e2) ->
      let r = freshIntName()  in
      let s = freshIntName()  in
      let e2CPS =
       cps_exp e2 ((*ClosedCPSCont*)(FnContCPS (s, BinOpAppCPS(k, b, r, s, ke)))) ke in
      cps_exp e1 ((*ClosedCPSCont*)(FnContCPS (r, e2CPS))) ke
(*[[if e1 then e2 else e3]]k,ke = [[e1]]((FN r -> if r then [[e2]]k,ke else [[e3]]k,ke),ke*)
   | IfExp (e1,e2,e3) ->
      let r =  freshIntName() in
      let e2cps = cps_exp e2 k ke  in
      let e3cps = cps_exp e3 k ke in
      cps_exp e1 ((*ClosedCPSCont*)(FnContCPS(r, IfCPS(r, e2cps, e3cps)))) ke
(*[[e1 e2]]k,ke = [[e1]](FN r -> [[e2]](FN s -> (r s k ke)),ke),ke*)
   | AppExp (e1,e2) ->
      let r = freshIntName() in
      let s = freshIntName() in
      let e2cps =
          cps_exp e2 ((*ClosedCPSCont*)(FnContCPS (s, AppCPS(k, r, s, ke)))) ke in
      cps_exp e1 ((*ClosedCPSCont*)(FnContCPS (r, e2cps))) ke
(*[[fun x -> e]]k,ke = k(FUN x -> fn kx kes-> [[e]]kx,kes) *)
   | FunExp (x,e) ->
     let (i,j) = (next_index(), next_index()) in
     let ecps = cps_exp e (ContVarCPS i) (ExnContVarCPS j) in
     FunCPS (k, x, i, j, ecps)
(*[[let x = e1 in e2)]]k,ke = [[e1]](FN x -> [[e2]]k,ke),ke *)
   | LetInExp (x,e1,e2) ->
     let e2cps = cps_exp e2 k ke in
     let fnk = FnContCPS(x,e2cps) in cps_exp e1 fnk ke
(*[[let rec f x = e1 in e2]]k,ke =
  (FN f -> [[e2]]k,ke)(FIX f. FUN x -> fn k',ke' => [[e1]]k',ke')*)
   | LetRecInExp(f,x,e1,e2) ->
     let (i,j) = (next_index(), next_index()) in
     let e1cps = cps_exp e1 (ContVarCPS i) (ExnContVarCPS j)  in
     let e2cps = cps_exp e2 k ke in
     let fnk = FnContCPS(f,e2cps) in
```

```
     FixCPS(fnk,f,x,i,j,e1cps)
(* [[try e with n0 -> e0 | ... | nm -> em]]k,ke =
   [[e]]k, [(n0 |-> [[e0]]k,ke); ... (nm |-> [[em]]k,ke)] + ke *)
   | TryWithExp(e, n0, e0, rem_match) ->
     let match_cps =
         List.fold_right
           (fun (n,en) -> fun exn_match ->
            let ecps = cps_exp en k ke in
                (n, ecps)::exn_match)
           ((n0, e0):: rem_match)
           []
     in cps_exp e k (UpdateExnContCPS(match_cps, ke))

(* [[raise e]]k,ke = [[e]](FN r -> match_exn r with ke), ke *)
   | RaiseExp e -> cps_exp e ((*ClosedCPSCont*)(ExnMatch ke)) ke

(*
and
   cps_dec dec ecps ke =
(*   <<val x = e>>ecps,ke = [[e]](FN x -> ecps),ke *)
   match dec with Val (x,e) -> cps_exp e ((*ClosedCPSCont*)(ContCPS (x, ecps))) ke
(*   <<dec1 dec2>>ecps, ke = <<dec1>>(<<dec2>>_ecps,ke),ke *)
   | Seq (dec1,dec2) -> let ecps2 =
     cps_dec dec2 ecps ke in cps_dec dec1 ecps2 ke
   | Local (dec1, dec2) ->  raise (Failure "Not implemented yet")
(*   <<val rec f = fn x => e>>_ecps = \mu x. [[e]]_FN f -> ecps *)
   | Rec (f,x,e) ->
     let (i,j) = (next_index(),next_index()) in
     let ecps2 = cps_exp e (ContVarCPS i) (ExnContVarCPS j) in
     FixCPS ((*ClosedCPSCont*)(ContCPS (f, ecps)), f, x, i, j, ecps2)
*)


let rec string_of_exp_cps ext_cps =
    match ext_cps
    with VarCPS (k,x) ->
        paren_string_of_cps_cont k ^ " " ^ x
    | ConstCPS (k,c) ->
        paren_string_of_cps_cont k ^ " " ^ string_of_const c
    | MonOpAppCPS (k,m,r, exncont) ->
        paren_string_of_cps_cont k ^ "(" ^  string_of_mon_op m ^ " " ^ r ^ ")"
    | BinOpAppCPS (k,b,r,s, exncont) ->
        paren_string_of_cps_cont k ^ "(" ^ r ^ " " ^ string_of_bin_op b ^ " " ^ s ^")"
    | IfCPS (b,e1,e2) ->
        "IF "^b^" THEN "^ string_of_exp_cps e1 ^" ELSE "^string_of_exp_cps e2
    | AppCPS (k,r,s, exncont) ->
        "("^r ^ " " ^ s ^ " " ^ paren_string_of_cps_cont k ^ ")"
    | FunCPS (k, x, i, j, e) ->
        (paren_string_of_cps_cont k) ^ " (" ^ (string_of_funk x e i j) ^ ")"
    | FixCPS (k,f,x,i,j, e) ->
        paren_string_of_cps_cont k ^ "(FIX "^ f ^". " ^ (string_of_funk x e i j) ^ ")"

and string_of_funk x e i j =
    "FUN " ^ x  ^ " -> " ^ "fn " ^ (string_of_int i) ^ ", " ^ (string_of_int j) ^ " => " ^
    string_of_exp_cps e

and
string_of_cps_cont k =
    match k
    with External -> "<external>"
    | ContVarCPS i -> "_k" ^ (string_of_int i)
    | FnContCPS (x, e) -> "FN " ^ x ^ " -> " ^ string_of_exp_cps e
    | ExnMatch exncont -> "<some exception>"

and

paren_string_of_cps_cont k =
    match k with FnContCPS _ -> "(" ^ string_of_cps_cont k ^ ")"
```

```
    | _ -> string_of_cps_cont k


let print_direct_result b =
    if b
        then print_string "Direct: Tail Recursive\n"
    else
        print_string "Direct: Not Tail Recursive\n"

let print_cps_result b =
    if b
        then print_string "CPS-transformed: Tail Recursive\n"
    else
        print_string "CPS-transformed: Not Tail Recursive\n"

let print_match d c =
    if not (d=c)
        then print_string "\n[!] Results don't match!\n\n";;
```

Listing 9: definitions.ml

5.4 tailRecPicoMLInt.ml

```
(*
 tailRecPicoMLInt.ml
 *)

open Definitions
open TailRecPicoMLparse
open TailRecPicoMLlex
open CheckTailRec
open CheckTailRecCPS

let is_interactive = true;;

(* default values *)
let is_cps_arg = ref false
let is_direct_arg = ref false


let usage = "usage: " ^ Sys.argv.(0) ^ " [-c] [-d]"

let speclist = [
    ("-c", Arg.Set is_cps_arg, ": CPS Only");
    ("-d", Arg.Set is_direct_arg, ": Direct style Only");
  ]

let _ =
    Arg.parse speclist (fun x -> raise (Arg.Bad ("Bad argument : " ^ x))) usage;
    (*Printf.printf " %b %b\n" !is_cps_arg !is_direct_arg;*)

    print_endline "\nWelcome to the PicoML Tail-resursion Checker \n";
    let rec loop gamma mem =
    try
        let lexbuf = Lexing.from_channel stdin
        in
        (print_string "> "; flush stdout);
        (
        try
            let dec = main
                (fun lb -> match token lb
                    with
                    | EOF -> raise EndInput
                    | r -> r
                ) lexbuf
            in
            match infer_dec gather_dec_ty_substitution gamma dec
            with
            | None ->
                (
                print_string "\ndoes not type check\n";
                loop gamma mem
                )
            | Some (Proof(hyps,judgement)) ->
                (

                let is_direct_true = (check_tail_recursion_direct dec ) in
                let is_cps_true = (check_tail_recursion_cps dec ) in

                let is_direct_only = ( (not !is_cps_arg) &&  !is_direct_arg ) in
                let is_cps_only = (!is_cps_arg && (not !is_direct_arg) ) in


                if is_direct_only
                    then (print_direct_result is_direct_true)
                else if is_cps_only
                    then (print_cps_result is_cps_true)
                else (* do both *)
```

```
                    ((print_direct_result is_direct_true);(print_cps_result is_cps_true);(
    print_match is_direct_true is_cps_true))
                ;
                loop gamma mem)
        with Failure s ->
            (
            print_newline();
            print_endline s;
            print_newline();
            loop gamma mem
            )
        | Parsing.Parse_error ->
            (
            print_string "\ndoes not parse\n";
            loop gamma mem
            )
        )
    with EndInput ->
        exit 0

in (loop [] []) ;;
```

Listing 10: tailRecPicoMLInt.ml

5.5 tailRecPicoMLTest.ml

```
(*
tailRecPicoMLTest.ml
 *)

open Definitions
open TailRecPicoMLparse
open TailRecPicoMLlex
open CheckTailRec
open CheckTailRecCPS

let check str =
    try
        let lexbuf = Lexing.from_string str
        in
        (
        try
            let dec = main
                (fun lb -> match token lb
                    with
                    | EOF -> raise EndInput
                    | r -> r
                ) lexbuf
            in
            match infer_dec gather_dec_ty_substitution [] dec
            with
            | None ->
                (
                print_string "\ndoes not type check\n"
                )
            | Some (Proof(hyps,judgement)) ->

                let is_direct_true = (check_tail_recursion_direct dec ) in
                let is_cps_true = (check_tail_recursion_cps dec ) in
                ((print_direct_result is_direct_true);(print_cps_result is_cps_true);(print_match
     is_direct_true is_cps_true))

        with Failure s ->
            (
            print_newline();
            print_endline s;
            print_newline()
            )
        | Parsing.Parse_error ->
            (
            print_string "\ndoes not parse\n";
            )
        )
    with EndInput ->
        exit 0
;;


let read_file filename =
    let lines = ref [] in
        let chan = open_in filename in
            try
                while true; do
                    let line = input_line chan
                    in
                    print_string line;
                    print_newline ();
                    check line ;
                    lines := line :: !lines;
                done; !lines
            with End_of_file ->
        close_in chan;
```

```
  List.rev !lines ;;

read_file "testing.txt";;
```

Listing 11: tailRecPicoMLTest.ml

5.6 tailRecPicoMLparse.mly

```
/* Use the expression datatype defined in expressions.ml: */
%{
    open Definitions
    let andsugar l r = IfExp(l,r,ConstExp (BoolConst false))
    let orsugar l r = IfExp(l,ConstExp (BoolConst true),r)
    let ltsugar l r = BinOpAppExp(GreaterOp,r,l)
    let leqsugar l r = orsugar (ltsugar l r) (BinOpAppExp(EqOp, l, r))
    let geqsugar l r = orsugar (BinOpAppExp(GreaterOp,l,r)) (BinOpAppExp(EqOp, l, r))
   (* let neqsugar l r = IfExp(BinOpAppExp (EqOp,l,r), ConstExp FalseConst,
                   ConstExp TrueConst) *)
    let neqsugar l r = BinOpAppExp(EqOp, BinOpAppExp (EqOp,l,r), ConstExp (BoolConst false))
%}

/* Define the tokens of the language: */
%token <int> INT
%token <float> FLOAT
%token <string> STRING IDENT
%token TRUE FALSE NEG PLUS MINUS TIMES DIV DPLUS DMINUS DTIMES DDIV MOD EXP CARAT
       LT GT LEQ GEQ EQUALS NEQ PIPE ARROW SEMI DSEMI DCOLON AT NIL
       LET REC AND IN IF THEN ELSE FUN MOD RAISE TRY WITH NOT LOGICALAND
       LOGICALOR LBRAC RBRAC LPAREN RPAREN COMMA UNDERSCORE UNIT
       HEAD TAIL PRINT FST SND EOF

/* Define the "goal" nonterminal of the grammar: */
%start main
%type <Definitions.dec> main

%%

main:
    expression DSEMI               { (Anon ( $1)) }
  | LET IDENT EQUALS expression DSEMI              { (Let ($2,$4)) }
  | LET REC IDENT IDENT EQUALS expression DSEMI    { (LetRec ($3, $4, $6)) }

expression:
   op_exp       { $1 }

op_exp:
  | pure_or_exp LOGICALOR and_exp { orsugar $1 $3 }
  | and_exp       { $1 }

and_exp:
  | pure_and_exp LOGICALAND rel_exp { andsugar $1 $3 }
  | rel_exp       { $1 }

rel_exp:
  | pure_rel_exp GT cons_exp    { BinOpAppExp (GreaterOp,$1,$3) }
  | pure_rel_exp EQUALS cons_exp  { BinOpAppExp (EqOp,$1,$3) }
  | pure_rel_exp LT cons_exp    { ltsugar $1 $3 }
  | pure_rel_exp LEQ cons_exp   { leqsugar $1 $3 }
  | pure_rel_exp GEQ cons_exp   { geqsugar $1 $3 }
  | pure_rel_exp NEQ cons_exp   { neqsugar $1 $3 }
  | cons_exp            { $1 }

cons_exp:
  | pure_add_exp DCOLON cons_exp  { BinOpAppExp(ConsOp,$1,$3) }
  | add_exp       { $1 }

add_exp:
  | pure_add_exp plus_minus mult_exp  { BinOpAppExp($2,$1,$3) }
  | mult_exp        { $1 }

mult_exp:
  | pure_mult_exp times_div expo_exp  { BinOpAppExp($2,$1,$3) }
  | expo_exp              { $1 }
```

```
expo_exp:
  | pure_app_raise_exp EXP expo_exp { BinOpAppExp (ExpoOp,$1,$3) }
  | nonop_exp               { $1 }

nonop_exp:
    if_let_fun_try_monop_exp      { $1 }
  | app_raise_exp     { $1 }

app_raise_exp:
    app_exp       { $1 }
  | monop_raise       { $1 }
  | pure_app_exp monop_raise    { AppExp($1,$2) }

monop_raise:
    monop RAISE nonop_exp   { MonOpAppExp ($1,RaiseExp($3)) }
  | RAISE nonop_exp     { RaiseExp $2 }

app_exp:
  | atomic_expression   { $1 }
  | pure_app_exp nonapp_exp   { AppExp($1,$2) }

nonapp_exp:
    atomic_expression   { $1 }
  | if_let_fun_try_monop_exp    { $1 }


if_let_fun_try_monop_exp:
    TRY expression WITH exp_matches { match $4 with (x,e,ms) -> TryWithExp ($2, x,e, ms) }
  | LET REC IDENT IDENT EQUALS expression IN expression { LetRecInExp($3, $4, $6, $8) }
  | LET IDENT EQUALS expression IN expression   { LetInExp($2, $4, $6) }
  | FUN IDENT ARROW expression        { FunExp($2, $4) }
  | IF expression THEN expression ELSE expression { IfExp($2, $4, $6) }
  | monop if_let_fun_try_monop_exp          { MonOpAppExp ($1,$2) }

exp_matches:
    exp_match         { (match $1 with (x,e) -> (x,e,[])) }
  | no_try_exp_match PIPE exp_matches   { (match ($1,$3) with (x,e),(y,f,l) -> (x,e,((y,f)::l)))
    }

exp_match:
    pat ARROW expression { ($1, $3) }

no_try_exp_match:
    pat ARROW no_try_expression   { ($1, $3) }


no_try_expression:
    no_try_op_exp     { $1 }

no_try_op_exp:
  | pure_or_exp LOGICALOR no_try_and_exp  { orsugar $1 $3 }
  | no_try_and_exp        { $1 }

no_try_and_exp:
    pure_and_exp LOGICALAND no_try_eq_exp { andsugar $1 $3 }
  | no_try_eq_exp         { $1 }

no_try_eq_exp:
  no_try_rel_exp          { $1 }

no_try_rel_exp:
  | pure_rel_exp GT no_try_cons_exp { BinOpAppExp (GreaterOp,$1,$3) }
  | pure_rel_exp EQUALS no_try_cons_exp { BinOpAppExp (EqOp,$1,$3) }
  | pure_rel_exp LT no_try_cons_exp { ltsugar $1 $3 }
  | pure_rel_exp GEQ no_try_cons_exp  { geqsugar $1 $3 }
  | pure_rel_exp LEQ no_try_cons_exp  { leqsugar $1 $3 }
  | pure_rel_exp NEQ no_try_cons_exp  { neqsugar $1 $3 }
  | no_try_cons_exp         { $1 }
```

```
no_try_cons_exp:
  | pure_add_exp DCOLON no_try_cons_exp { BinOpAppExp(ConsOp,$1,$3) }
  | no_try_add_exp      { $1 }

no_try_add_exp:
  | pure_add_exp plus_minus no_try_mult_exp { BinOpAppExp($2,$1,$3) }
  | no_try_mult_exp       { $1 }

no_try_mult_exp:
  | pure_mult_exp times_div no_try_expo_exp { BinOpAppExp(IntTimesOp,$1,$3) }
  | no_try_expo_exp       { $1 }

no_try_expo_exp:
  | pure_app_raise_exp EXP no_try_expo_exp  { BinOpAppExp(ExpoOp,$1,$3) }
  | no_try_nonop_exp                       { $1 }

no_try_nonop_exp:
    no_try_if_let_fun_monop_exp   { $1 }
  | no_try_app_raise_expression   { $1 }

no_try_app_raise_expression:
    no_try_app_expression   { $1 }
  | no_try_monop_expression   { $1 }
  | pure_app_exp no_try_monop_expression  { $1 }

no_try_monop_expression:
  | monop RAISE no_try_app_raise_expression { MonOpAppExp($1,RaiseExp($3)) }
  | RAISE no_try_app_raise_expression  { RaiseExp($2) }

no_try_app_expression:
    atomic_expression       { $1 }
  | pure_app_exp no_try_nonapp_expression   { AppExp($1,$2) }

no_try_nonapp_expression:
    atomic_expression     { $1 }
  | no_try_if_let_fun_monop_exp   { $1 }

no_try_if_let_fun_monop_exp:
    IF expression THEN expression ELSE no_try_expression  { IfExp($2,$4,$6) }
  | LET IDENT EQUALS expression IN no_try_expression    { LetInExp($2,$4,$6) }
  | LET REC IDENT IDENT EQUALS expression IN no_try_expression  { LetRecInExp($3,$4,$6,$8) }
  | FUN IDENT ARROW no_try_expression       { FunExp($2, $4) }
  | monop no_try_if_let_fun_monop_exp       { MonOpAppExp ($1,$2) }

pat:
  | UNDERSCORE  { None }
  | INT   { Some $1 }

pure_or_exp:
  | pure_or_exp LOGICALOR pure_and_exp    { orsugar $1 $3 }
  | pure_and_exp       { $1 }

pure_and_exp:
  | pure_and_exp LOGICALAND pure_eq_exp { andsugar $1 $3 }
  | pure_eq_exp           { $1 }

pure_eq_exp:
  pure_rel_exp          { $1 }

pure_rel_exp:
  | pure_rel_exp GT pure_cons_exp { BinOpAppExp (GreaterOp,$1,$3) }
  | pure_rel_exp EQUALS pure_cons_exp { BinOpAppExp (EqOp,$1,$3) }
  | pure_rel_exp LT pure_cons_exp { ltsugar $1 $3 }
  | pure_rel_exp GEQ pure_cons_exp  { geqsugar $1 $3 }
  | pure_rel_exp LEQ pure_cons_exp  { leqsugar $1 $3 }
  | pure_rel_exp NEQ pure_cons_exp  { neqsugar $1 $3 }
  | pure_cons_exp         { $1 }
```

```

pure_cons_exp:
  | pure_add_exp DCOLON pure_cons_exp   { BinOpAppExp(ConsOp,$1,$3) }
  | pure_add_exp      { $1 }

pure_add_exp:
  | pure_add_exp plus_minus pure_mult_exp { BinOpAppExp($2,$1,$3) }
  | pure_mult_exp       { $1 }

pure_mult_exp:
  | pure_mult_exp times_div pure_expo_exp   { BinOpAppExp($2,$1,$3) }
  | pure_expo_exp             { $1 }

pure_expo_exp:
  | pure_app_raise_exp EXP pure_expo_exp  { BinOpAppExp (ExpoOp,$1,$3) }
  | pure_app_raise_exp           { $1 }

pure_app_raise_exp:
    pure_app_exp    { $1 }
  | pure_monop_raise    { $1 }
  | pure_app_exp pure_monop_raise { AppExp($1,$2) }

pure_monop_raise:
    monop RAISE pure_app_raise_exp { MonOpAppExp($1,RaiseExp($3)) }
  | RAISE pure_app_raise_exp  { RaiseExp($2) }

pure_app_exp:
    atomic_expression     { $1 }
  | pure_app_exp atomic_expression  { AppExp($1,$2) }

atomic_expression:
    constant_expression         { ConstExp $1 }
  | IDENT     { VarExp $1 }
  | list_expression   { $1 }
  | paren_expression            { $1 }
  | monop atomic_expression   { MonOpAppExp ($1,$2) }

list_expression:
    LBRAC list_contents     { $2 }

list_exp_end:
    RBRAC       { ConstExp NilConst }
  | SEMI list_tail        { $2 }

list_tail:
    RBRAC       { ConstExp NilConst }
  | list_contents     { $1 }

list_contents:
    expression list_exp_end { BinOpAppExp(ConsOp,$1,$2) }

paren_expression:
    LPAREN par_exp_end      { $2 }

par_exp_end:
    RPAREN                { ConstExp UnitConst }
  | expression RPAREN     { $1 }
  | expression COMMA expression RPAREN  { BinOpAppExp (CommaOp,$1,$3) }

constant_expression:
    INT                         { IntConst $1 }
  | TRUE      { BoolConst true }
  | FALSE     { BoolConst false }
  | FLOAT     { FloatConst $1 }
  | NIL         { NilConst }
  | STRING      { StringConst $1 }
  | UNIT      { UnitConst }

```

```

monop:
  | HEAD      { HdOp }
  | TAIL      { TlOp }
  | PRINT     { PrintStringOp }
  | NEG       { IntNegOp }
  | FST       { FstOp }
  | SND       { SndOp }

plus_minus:
    PLUS        { IntPlusOp }
  | MINUS       { IntMinusOp }
  | DPLUS       { FloatPlusOp }
  | DMINUS        { FloatMinusOp }
  | CARAT       { ConcatOp }

times_div:
    TIMES       { IntTimesOp }
  | DIV         { IntDivOp }
  | MOD         { ModOp }
  | DTIMES        { FloatTimesOp }
  | DDIV        { FloatDivOp }
```

Listing 12: tailRecPicoMLparse.mly

5.7 tailRecPicoMLlex.ml

```
{
open Definitions;;
open TailRecPicoMLparse;;

exception EndInput

}

(* You can assign names to commonly-used regular expressions in this part
   of the code, to save the trouble of re-typing them each time they are used *)

let numeric = ['0' - '9']
let lowercase = ['a' - 'z']
let letter =['a' - 'z' 'A' - 'Z' '_']
let hex = ['0' - '9' 'a' - 'f']
let ident_char = letter | numeric | '_' | '\''
let string_char = ident_char | ' ' | '~' | '\\' | '!' | '@' | '#' | '$' | '%' | '^' | '&'
  | '*' | '(' | ')' | '-' | '+' | '=' | '{' | '[' | '}' | ']'
  | '|' | ':' | ';' | '<' | ',' | '>' | '.' | '?' | '/'

      rule token = parse
        | [' ' '\t' '\n'] { token lexbuf }  (* skip over whitespace *)
        | eof             { EOF }
          (* binary operators *)
        | "+"    { PLUS }
        | "-"    { MINUS }
        | "*"    { TIMES }
        | "/"    { DIV }
        | "+."   { DPLUS }
        | "-."   { DMINUS }
        | "*."   { DTIMES }
        | "/."   { DDIV }
        | "^"    { CARAT }
        | "::"   { DCOLON }
        | "<"    { LT }
        | ">"    { GT }
        | "="    { EQUALS }
        | ">="   { GEQ }
        | "<="   { LEQ }
        | "<>"   { NEQ }
        | "mod"  { MOD }
        | "**"   { EXP }
          (* monadic operators *)
        | "fst"          { FST }
        | "snd"          { SND }
        | "hd"           { HEAD }
        | "tl"           { TAIL }
        | "print_string" { PRINT }
        | "~"            { NEG }
          (* top-level/let-exp keywords *)
        | "let"    { LET }
        | "rec"    { REC }
        | "in"     { IN }
        | ";;"     { DSEMI }
          (* tuple/list symbols *)
        | "("      { LPAREN }
        | ")"      { RPAREN }
        | ","      { COMMA }
        | "["      { LBRAC }
        | "]"      { RBRAC }
        | ";"      { SEMI }
          (* boolean operators *)
        | "&&"     { LOGICALAND }
        | "||"     { LOGICALOR }
          (* if-then-else keywords *)
        | "if"     { IF }
```

```
        | "then"  { THEN }
        | "else"  { ELSE }
          (* function keywords *)
        | "fun"   { FUN }
        | "->"    { ARROW }
          (* exception handling keywords *)
        | "raise" { RAISE }
        | "try"   { TRY }
        | "with"  { WITH }
        | "|"     { PIPE }
        | "_"     { UNDERSCORE }
          (* named constants *)
        | "true"  { TRUE }
        | "false" { FALSE }
        | "[]"    { NIL }
        | "()"    { UNIT }
          (* numeric constants *)
        | numeric+ as s { INT (int_of_string s) }
        | numeric+'.'(numeric*) as s { FLOAT (float_of_string s) }
        | "0b"(('0'|'1')+) as s { INT (int_of_string s) }
        | "0x"(hex+) as s { INT (int_of_string s) }
        | numeric+'.'(numeric*)'e'(numeric+) as s   { FLOAT (float_of_string s) }
          (* identifiers *)
        | lowercase (ident_char*) as s { IDENT s }
          (* string literals *)
        | "\""   { string "" lexbuf }
      and string ins = parse
        | string_char* as s { string (ins ^ s) lexbuf }
        | "\""  { STRING ins }
        | "\\\\" {string (ins ^ "\\") lexbuf }
        | "\\\'" {string (ins ^ "\'") lexbuf }
        | "\\\"" {string (ins ^ "\"") lexbuf }
        | "\\t" {string (ins ^ "\t") lexbuf }
        | "\\n" {string (ins ^ "\n") lexbuf }
        | "\\r" {string (ins ^ "\r") lexbuf }
        | "\\b" {string (ins ^ "\b") lexbuf }
        | "\\\ " {string (ins ^ "\ ") lexbuf }
        | "\\"(('0'|'1')numeric numeric as s )
            {string (ins ^ String.make 1 (char_of_int (int_of_string s))) lexbuf }
        | "\\"('2'['0' - '4']numeric as s)
            {string (ins ^ String.make 1 (char_of_int (int_of_string s))) lexbuf }
        | "\\"("25"['0' - '5'] as s)
            {string (ins ^ String.make 1 (char_of_int (int_of_string s))) lexbuf }

            (* your rules go here *)


            {(* do not modify this function: *)
              let lextest s = token (Lexing.from_string s)

 let get_all_tokens s =
     let b = Lexing.from_string (s^"\n") in
     let rec g () =
     match token b with EOF -> []
     | t -> t :: g () in
     g ()

let try_get_all_tokens s =
    try (Some (get_all_tokens s), true)
    with Failure "unmatched open comment" -> (None, true)
       | Failure "unmatched closed comment" -> (None, false)

let get_all_token_options s =
  let b = Lexing.from_string (s^"\n") in
  let rec g () =
    match (try Some (token b) with _ -> None) with Some EOF -> []
      | None -> [None]
      | t -> t :: g () in
```

```
  g ()

}
```

Listing 13: tailRecPicoMLlex.mll

References

[1] Robert I. Soare, *Computability and recursion*, BULL. SYMBOLIC LOGIC, 1996

[2] Programming Languages and Compilers : CS421, Fall 2015, University of Illinois, Urbana Champaign, Course Web Page