Appendix A: Interpreter Code

1 type var = string
2
type exp =
4 | Var of var
5 | Imm of int
6 | Add of exp * exp
7 | Mul of exp * exp
8
type cmd =
10 | Skip
11 | Assn of var * exp
12 | Seq of cmd * cmd
13 | IfNZ of exp * cmd * cmd
14 | WhileNZ of exp * cmd
15 | For of var * exp * cmd (* <------- This is the new construct *)
16
type state = (var * int) list
18
let set (s:state) (x:var) (v:int) =
20 (x,v)::s
22
let rec get (s:state) (x:var) : int =
24 begin match s with
25 | [] -> 0 (* uninitialized variables are 0 *)
26 | (y,v)::rest -> if x = y then v else get rest x
27 end
29
let rec interp_exp (e:exp) (s:state) : int =
31 begin match e with
32 | Var x -> get s x
33 | Imm v -> v
34 | Add(e1, e2) -> (interp_exp e1 s) + (interp_exp e2 s)
35 | Mul(e1, e2) -> (interp_exp e1 s) * (interp_exp e2 s)
36 end
let rec interp_cmd (c:cmd) (s:state) : state =
    begin match c with
        | Skip -> s
        | Assn(x, e) -> set s x (interp_exp e s)
        | Seq(c1, c2) -> interp_cmd c2 (interp_cmd c1 s)
        | IfNZ(e, c1, c2) ->
            interp_cmd (if (interp_exp e s) <> 0 then c1 else c2) s
        | WhileNZ(e, c) ->
            interp_cmd (IfNZ(e, Seq(c, WhileNZ(e, c)), Skip)) s
        | For(x, e, c) ->
            let s0 = set s x 0 in
            let rec loop s =
                let e = interp_exp e s in
                let v = get s x in
                if v = e then s else
                    let s' = interp_cmd c s in
                    let v' = get s' x in
                    loop (set s' x (v'+1))
            in
            loop s0
    end

(* The cmd [factorial_for] computes factorial of 6 using a for loop
 (and the available SIMPLE arithmetic instructions): *)

let factorial_for : cmd =
    let x = "x" in
    let ans = "ans" in
    Seq(Assn(x, Imm 6),
        Seq(Assn(ans, Imm 1),
            For("y", Var x,
                Assn(ans, Mul(Var ans, (Add(Var x, Mul(Var "y", Imm(-1)))))
            )))
    )
Appendix B: Lexer Code

```ocaml
{ open Lexing

type token = | Int of int64 | Ident of string | LPAREN |
| LPARENSTAR | STARRPAREN | IF

let print_token t =
  begin match t with
  | Int x -> (Printf.printf "Int %Ld\n%!" x)
  | Ident s -> (Printf.printf "Ident %s\n%!" s)
  | IF -> (Printf.printf "IF\n%!"
  | LPAREN -> (Printf.printf "LPAREN\n%!"
  | LPARENSTAR -> (Printf.printf "LPARENSTAR\n%!"
  | STARRPAREN -> (Printf.printf "STARRPAREN\n%!"

end

let acc = ref []

let emit t = acc := t::(!acc)

exception Lex_error of char

let character = ['a'-'z''A'-'Z']

let digit = ['0'-'9']

let underscore = [ '_ ']

let whitespace = [' ' '	' '
' '']

let identifier = character (character|digit|underscore)*

rule lex = parse

| "if" { emit IF; lex lexbuf }
| identifier { emit (Ident (lexeme lexbuf)); lex lexbuf }
| '(' { emit LPAREN; lex lexbuf }
| "(" { emit LPARENSTAR; lex lexbuf }
| "*" { emit STARRPAREN; lex lexbuf }
| whitespace+ { lex lexbuf }
| digit+ { emit (Int (Int64.of_string (lexeme lexbuf))); lex lexbuf }
| eof { List.rev (!acc) }
| _ as c { raise (Lex_error c) }

try
  List.iter print_token (lex (from_channel stdin))
with
  Lex_error c -> Printf.printf "Char %s is unexpected.\n" (Char.escaped c)
}
```