open Ast

let *from\_some* **=** function

**|** **None** **->** failwith "from\_some: None"

**|** **Some** *v* **->** v

type **tenv** **=**

**|** **EmptyTEnv**

**|** **ExtendTEnv** of string**\***texpr**\***tenv

let *empty\_tenv* **()** **=** **EmptyTEnv**

let *extend\_tenv* *id* *t* *tenv* **=** **ExtendTEnv**(id,t,tenv)

let rec *apply\_tenv* (*tenv***:**tenv) (*id***:**string)**:**texpr option =

match tenv with

**|** EmptyTEnv **->** None

**|** ExtendTEnv (key,value,tenv1) **->** if id**=**key

then **Some** value

else apply\_tenv tenv1 id

let rec *string\_of\_tenv* **=** function

**|** **EmptyTEnv** **->** ""

**|** **ExtendTEnv**(*id***,***v***,***env*) **->** "("^id^","^string\_of\_texpr v^")"^string\_of\_tenv env

let rec *type\_of\_prog* **=** function

**|** **AProg** *e* **->** type\_of\_expr (init\_tenv **()**) e

and

*type\_of\_expr* *en* **=** function

**|** **Int** *n* **->** **IntType**

**|** **Var** *id* **->**

(match apply\_tenv en id with

**|** **None** **->** failwith @@ "Variable "^id^" undefined"

**|** **Some** *texp* **->** texp)

**|** **Unit** **->** **UnitType**

**|** **ITE**(*e1***,** *e2***,** *e3*) **->**

let *t1* **=** type\_of\_expr en e1 in let *t2* **=** type\_of\_expr en e2 in let *t3* **=** type\_of\_expr en e3 in if t1=**BoolType** && t2=t3

then t2

else failwith "ITE: Type error"

**|** **Add**(*e1***,** *e2*) **|** **Mul**(*e1***,***e2*) **|** **Sub**(*e1***,***e2*) **|** **Div**(*e1***,***e2*) **->**

let *t1* **=** type\_of\_expr en e1 in let *t2* **=** type\_of\_expr en e2 in if t1=**IntType** && t2=**IntType**

then **IntType**

else failwith "Add: arguments must be ints"

**|** **IsZero**(*e*) **->**

let *t1* **=** type\_of\_expr en e in if t1=**IntType**

then **BoolType**

else failwith "Zero?: argument must be int"

**|** **Let**(*x***,** *e1***,** *e2*) **->**

let *t1* **=** type\_of\_expr en e1

in type\_of\_expr (extend\_tenv x t1 en) e2

**|** **Proc**(*x***,***ty***,***e*) **->**

let *tc***=** type\_of\_expr (extend\_tenv x ty en) e

in **FuncType**(ty,tc)

**|** **App**(*e1***,***e2*) **->**

let *t1* **=** type\_of\_expr en e1 in let *t2* **=** type\_of\_expr en e2 in (match t1 with

**|** **FuncType**(*td***,***tcd*) when td=t2 **->** tcd

**|** **FuncType**(*td***,***tcd*) **->** failwith "App: argument does not have correct type"

**|** \_ **->** failwith "Checker: App: LHS must be function type")

**|** **Letrec**(*tRes***,***id***,***param***,***tParam***,***body***,***e*) **->** let *t***=**type\_of\_expr (extend\_tenv param tParam (extend\_tenv id (**FuncType**(tParam,tRes)) en)) body in if t=tRes

then type\_of\_expr (extend\_tenv id (**FuncType**(tParam,tRes)) en) e

else failwith "Checker: LetRec: Types of recursive function does not match declaration"

**|** **Set**(*id***,***e*) **->**

failwith "EXPLICIT-REFS: Set not a valid operation"

**|** **BeginEnd**(*es*) **->**

List**.**fold\_left (fun *v* *e* **->** type\_of\_expr en e) **UnitType** es

(\* *explicit ref* \*)

**|** **NewRef**(*e*) **->**

let *t* **=** type\_of\_expr en e in

**RefType**(t)

**|** **DeRef**(*e*) **->**

let *t* **=** type\_of\_expr en e in

(match t with

**|** **RefType**(*tx*) **->** tx

**|** \_ **->** failwith "DeRef: argument must be RefType")

**|** **SetRef**(*e1***,***e2*) **->**

let *t1* **=** type\_of\_expr en e1 in let *t2* **=** type\_of\_expr en e2 in (match t2 with

**|** **RefType**(*tx*) **->**

if tx=t1

then **UnitType**

else failwith "SetRef: type of first argument does not match second argument"

**|** \_ **->** failwith "SetRef: arguments must be RefType")

(\* *pair* \*)

**|** **Pair**(*e1***,** *e2*) **->**

let *t1* **=** type\_of\_expr en e1 in let *t2* **=** type\_of\_expr en e2 in **PairType**(t1, t2)

**|** **Unpair**(*id1***,** *id2***,** *def***,** *body*) **->** let *t* **=** type\_of\_expr en def in (match t with

**|** **PairType**(*e1***,** *e2*) **->**

type\_of\_expr (extend\_tenv id1 e1 (extend\_tenv id2 e2 en)) body

**|** \_ **->** failwith "Unpair: arguments must be PairType")

(\* *list* \*)

**|** **EmptyList**(*t*) **->** **ListType**(t)

**|** **Cons**(*he***,** *te*) **->** let *t1* **=** type\_of\_expr en he in let *t2* **=** type\_of\_expr en te in

(match t2 with

**|** **ListType**(*tx*) **->** if tx=t1 then **ListType**(t1) else failwith "Cons: type of first argument does not match second argument"

**|** \_ **->** failwith "Cons: arguments must be ListType")

**|** **Null**(*e*) **->**

let *t* **=** type\_of\_expr en e in

(match t with

**|** **ListType**(*tx*) **->** **BoolType**

**|** \_ **->** failwith "Null: argument must be ListType")

**|** **Hd**(*e*) **->**

let *t* **=** type\_of\_expr en e in

(match t with

**|** **ListType**(*tx*) **->** tx

**|** \_ **->** failwith "Hd: argument must be ListType")

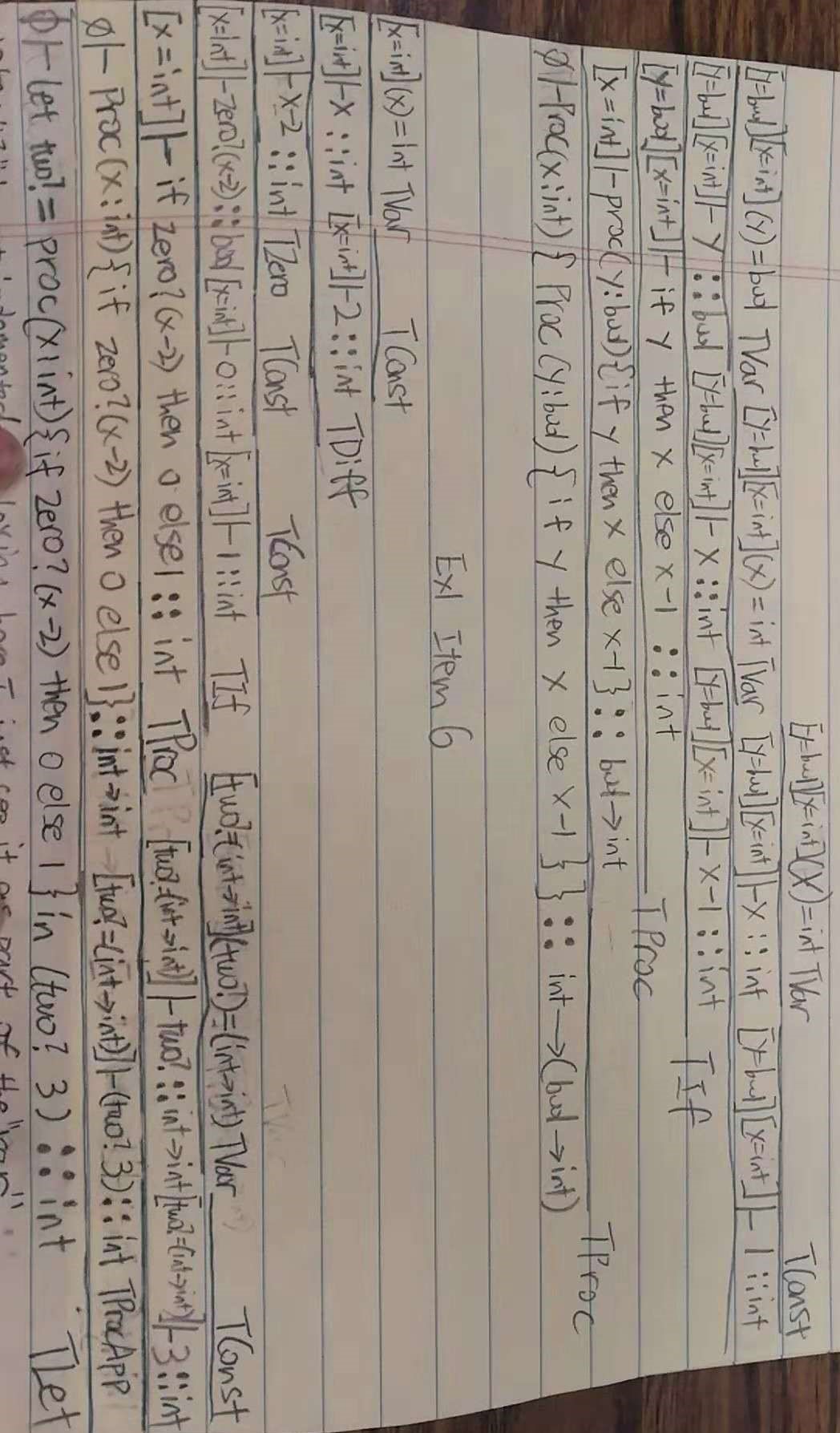
**|** **Tl**(*e*) **->**

let *t* **=** type\_of\_expr en e in

(match t with

**|** **ListType**(*tx*) **->** t

**|** \_ **->** failwith "Tl: argument must be ListType")

 (\* *tree* \*)

**|** **EmptyTree**(*t*) **->** **TreeType**(t)

**|** **Node**(*de***,** *le***,** *re*) **->**

let *t1* **=** type\_of\_expr en de in let *t2* **=** type\_of\_expr en le in let *t3* **=** type\_of\_expr en re in (match t2, t3 with

**|** **TreeType**(*tx*)**,TreeType**(*ty*) **->**

if tx = ty

then

(if t1 = tx

then t2

else failwith "Node: type of first argument does not match second argument")

else failwith "Node: type of second argument does not match third argument"

**|** \_ **->** failwith "Node: arguments must be TreeType")

**|** **NullT**(*t*) **->**

let *tn* **=** type\_of\_expr en t in (match tn with

**|** **TreeType**(*tx*) **->** **BoolType**

**|** \_ **->** failwith "NullT: argument must be TreeType")

**|** **GetData**(*t*) **->**

let *tg* **=** type\_of\_expr en t in (match tg with

**|** **TreeType**(*tx*) **->** tx

**|** \_ **->** failwith "GetData: argument must be TreeType")

**|** **GetLST**(*t*) **->**

let *tl* **=** type\_of\_expr en t in (match tl with

**|** **TreeType**(*tx*) **->** tl

**|** \_ **->** failwith "GetLST: argument must be TreeType")

**|** **GetRST**(*t*) **->**

let *tr* **=** type\_of\_expr en t in (match tr with

**|** **TreeType**(*tx*) **->** tr

**|** \_ **->** failwith "GetRST: argument must be TreeType")

**|** **Debug** **->**

print\_string "Environment:\n";

print\_string @@ string\_of\_tenv en;

**UnitType**

let *parse* *s* **=**

let *lexbuf* **=** Lexing**.**from\_string s in

let *ast* **=** Parser**.**prog Lexer**.**read lexbuf in ast

(\* *Interpret an expression* \*)

let *chk* (*e***:**string) **:** texpr **=**

e |> parse |> type\_of\_prog