```
let rec type_of_expr expr env =
let aux expr env =
if ( ! debug ) then
(print_endline ((string_of_ast expr) ^ " -> "
^ (string_of_type_env env)));
match expr with
(* Partie Expression *)
| (TrueNode) -> ruleBoolean
| (FalseNode) -> ruleBoolean
| (IntegerNode _) -> ruleInteger
| (AccessNode name) -> ruleAccess env
name
| (BinaryNode (op,left,right)) ->
ruleBinary env op left right
| (UnaryNode (op,subexpr)) -> ruleUnary
env op subexpr

(* Partie Fonctionnelle *)
| (LetNode (ident,bvalue,bin)) -> ruleLet
env ident bvalue bin
| (IfthenelseNode (cond, ethen, eelse)) -
> ruleIf env cond ethen eelse
| (FunctionNode (par,body)) ->
ruleFunction env par body
| (CallNode (fct,par)) -> ruleCall env fct
par
| (LetrecNode (ident,bvalue,bin)) ->
ruleLetrec env ident bvalue bin

(* Partie Impérative *)
| (UnitNode) -> ruleUnit
| (RefNode subexpr) -> ruleRef env
subexpr
| (ReadNode subexpr) -> ruleRead env
subexpr
| (WriteNode (left,right)) -> ruleWrite env
left right
| (SequenceNode (left,right)) ->
ruleSequence env left right
| (WhileNode (cond,body)) -> ruleWhile
env cond body
in
(normalize (aux expr env))

and
(* ..... .*)
ruleBoolean = BooleanType

and
(* ..... .*)
ruleInteger = IntegerType

and
(* ..... .*)
ruleAccess env name =
(match (lookforEnv name env) with
| NotFound -> ErrorType
| (Found t) -> t)

and
(* ..... .*)
ruleUnary env op expr =
(match op with
| Negation ->
let texpr = (type_of_expr expr env) in
let _, ure = unify texpr BooleanType in
(if (ure) then BooleanType else
ErrorType)
| Opposite ->
let texpr = (type_of_expr expr env) in
let _, ure = unify texpr IntegerType in
(if (ure) then IntegerType else ErrorType)
(* | _ -> ErrorType *)
)

and
(* ..... .*)
ruleBinary env op left right =
let tleft = (type_of_expr left env) in
let tright = (type_of_expr right env) in
(match op with
| (Equal | Different | Lesser |
LesserEqual | Greater | GreaterEqual) -
>
let _, url = unify tleft IntegerType in
let _, urr = unify tright IntegerType in
(if (url && urr) then BooleanType else
ErrorType)
| (Add | Substract | Multiply | Divide) ->
let _, url = unify tleft IntegerType in
let _, urr = unify tright IntegerType in
(if (url && urr) then IntegerType else
ErrorType)
| (Or | And) ->
let _, url = unify tleft BooleanType in
```

```
let _ , urr = unify tright BooleanType in
(if (url && urr) then BooleanType else
ErrorType)
(*
| _ -> ErrorType)
*)
)

and
(* ..................................................................
.*)
ruleLet env ident bvalue bin =
let typeident = (type_of_expr bvalue env)
in
(* (print_endline ((string_of_ast bvalue) ^ "
-> " ^ (string_of_type typeident)));
(print_endline ((string_of_ast bin))); *)
(type_of_expr bin ((ident,typeident)::env))

and
(* ..................................................................
.*)
ruleIf env cond ethen eelse =
let tcond = (type_of_expr cond env) in
let tthen = (type_of_expr ethen env) in
let telse = (type_of_expr eelse env) in
let _ ,urc = unify tcond BooleanType in
let ut,ur = unify tthen telse in
(if (urc && ur) then ut else ErrorType)

and
(* ..............A
COMPLETER .....................................*)
ruleFunction env par body =
let t1 = newVariable() in
let tbody = (type_of_expr body
((par,t1)::env)) in
FunctionType(t1,tbody)
```