1 O'Caml

1.1 Scanner

```
{ open Parser }

let letter = ['a' - 'z' 'A' - 'Z']
let digit = ['0' - '9']
let quote = '"'

rule token = parse
  [' ' '\r' '\n' '\t'] { token lexbuf } | "/*"    { comment lexbuf }
  | '('     { LPAREN }    | ')'     { RPAREN }
  | '{'     { LBRACE }    | '}'     { RBRACE }    | ','     { COMMA }
  | '+'     { PLUS }    | '-'     { MINUS }
  | '*'     { TIMES }   | '/'     { DIVIDE }
  | '='     { ASSIGN }    | ';'     { SEMI }
  | "<<-"     { MOVE }
  | "<-"      { COPY }
  | "=="      { EQ }      | "!="      { NEQ }
  | '<'     { LT }      | "<="      { LEQ }
  | '>'     { GT }      | ">="      { GEQ }
  | '['     { LBRACK }    | ']'     { RBRACK }
  | "&&"      { AND }     | "||"      { OR }
  | '!'     { NOT }     | ".name"   { PATHNAME }
  | "def"     { DEF }     | ".created_at" { PATHCREATED }
  | "int"     { INT }     | ".kind"   { PATHKIND }
  | "path"    { PATH }
  | "string"    { STR }     | "list"    { LIST }
  | "if"      { IF }      | "else"    { ELSE }
  | "then"    { THEN }    | "print"   { PRINT }
  | "for"     { FOR }     | "in"      { IN }
  | "do"      { DO }      | "bool"    { BOOL }
  | "while"   { WHILE }   | "return"    { RETURN }
  | "void"    { VOID }    | ".add"    { ADD }
  | "true"    { TRUE }    | ".remove"   { REMOVE }
  | "false"   { FALSE }   | ".type"   { PATHEXT }
  | "trash"   { TRASH }
  | eof     { EOF }     (* do as microC *)
  | digit+ as lit         { LIT_INT(int_of_string lit) }
  | quote [^'"']* quote as lit  { LIT_STR(lit) }
  | letter | (letter | digit | '_')* as id    { ID(id) }
  | _ as char     { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
  "*/"      { token lexbuf }
  | _       { comment lexbuf}
```

1.2 Parser

```
%{ open Ast %}

%token LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK COMMA TAB SEMI
%token PLUS MINUS TIMES DIVIDE ASSIGN MOVE COPY
%token EQ NEQ LT LEQ GT GEQ NOT
%token AND OR
%token RETURN IF THEN ELSE FOR IN WHILE DO
%token DEF VOID INT STR LIST PATH BOOL TRASH TRUE FALSE PRINT
%token PATHNAME PATHCREATED PATHKIND PATHEXT ADD REMOVE
%token <int> LIT_INT
%token <string> LIT_STR
%token <bool> LIT_BOOL
%token <string> ID
%token IN
%token EOF

%nonassoc NOELSE
%nonassoc ELSE

%right ASSIGN MOVE COPY NOT

%left AND OR
%left EQ NEQ
%left LT GT LEQ GEQ
%left IN
%left PLUS MINUS
%left TIMES DIVIDE

%start program
%type <Ast.program> program

%%

program:
    { [], [] }
    | program vdecl { ($2 :: fst $1), snd $1 }
    | program fdecl { fst $1, ($2 :: snd $1) }

fdecl:
    DEF return_type ID LPAREN formals_opt RPAREN LBRACE vdecl_opt stmt_list RBRACE
       {{
        return = $2;
        fname = $3;
        formals = $5;
        fnlocals = List.rev $8;
        body = List.rev $9 }}
```

```
return_type:
      VOID      { VoidType }
    | INT       { IntType }
    | BOOL      { BoolType }
    | PATH      { PathType }
    | STR       { StrType }
    | LIST      { ListType }

formals_opt:
    { [] }
    | formal_list   { List.rev $1 }

formal_list:
    formal                    { [$1] }
    | formal_list COMMA formal { $3 :: $1 }

formal:
    INT ID        { { vtype = IntType;  vname = $2; vexpr = Noexpr; } }
    | BOOL ID     { { vtype = BoolType; vname = $2; vexpr = Noexpr; } }
    | PATH ID     { { vtype = PathType; vname = $2; vexpr = Noexpr; } }
    | STR ID      { { vtype = StrType;  vname = $2; vexpr = Noexpr; } }
    | LIST ID     { { vtype = ListType; vname = $2; vexpr = Noexpr; } }

vdecl_opt:
    { [] }
    | vdecl_list    { List.rev $1 }

vdecl_list:
    vdecl             { [$1] }
    | vdecl_list vdecl { $2 :: $1 }

vdecl:
      vdecl_type ID SEMI    { { vtype = $1;  vname = $2; vexpr = Noexpr } }
    | vdecl_type ID ASSIGN expr SEMI  { { vtype = $1;  vname = $2; vexpr = $4 } }

vdecl_type:
    VOID            { VoidType }
    | INT           { IntType }
    | BOOL          { BoolType }
    | STR           { StrType }
    | PATH          { PathType }
    | LIST          { ListType }

stmt_list:
    { [] }
    | stmt_list stmt { $2 :: $1 }

rev_stmt_list:
    stmt_list          { List.rev $1 }

stmt:
    expr SEMI                                          { Expr($1) }
    | RETURN expr_opt SEMI                             { Return($2) }
    | IF LPAREN expr RPAREN THEN stmt %prec NOELSE     { If($3, $6, Block([])) }
    | IF LPAREN expr RPAREN THEN stmt ELSE stmt        { If($3, $6, $8) }
    | PRINT expr SEMI                                  { Print($2) }
    | WHILE LPAREN expr RPAREN stmt                    { While($3, $5) }
    | FOR LPAREN for_expr IN for_expr RPAREN stmt      { For($3, $5, $7 ) }
    | IF list_expr IN list_expr THEN stmt %prec NOELSE { Ifin($2, $4, $6, Block([])) }
    | IF list_expr IN list_expr THEN stmt ELSE stmt    { Ifin($2, $4, $6, $8) }
    | LBRACE rev_stmt_list RBRACE                      { Block($2) }


for_expr:
    ID                             { Forid($1) }

list_expr:
    ID                            { ListId($1) }
    | LIT_INT                     { ListItemInt($1) }
    | LIT_STR                     { ListItemStr($1) }

expr_opt:
    /* nothing */ { Noexpr }
  | expr          { $1 }
```

```
expr:
    | LIT_INT                      { LitInt($1) }
    | TRUE                         { LitInt(1) }
    | FALSE                        { LitInt(0) }
    | LIT_STR                      { LitStr($1) }
    | LBRACK list_items RBRACK     { List($2) }
    | ID                           { Id($1) }
    | expr PLUS   expr             { Binop($1, Add,      $3) }
    | expr MINUS  expr             { Binop($1, Sub,      $3) }
    | expr TIMES  expr             { Binop($1, Mult,     $3) }
    | expr DIVIDE expr             { Binop($1, Div,      $3) }
    | expr EQ     expr             { Binop($1, Equal,    $3) }
    | expr NEQ    expr             { Binop($1, Neq,      $3) }
    | expr LT     expr             { Binop($1, Less,     $3) }
    | expr LEQ    expr             { Binop($1, Leq,      $3) }
    | expr GT     expr             { Binop($1, Greater,  $3) }
    | expr GEQ    expr             { Binop($1, Geq,      $3) }
    | expr AND expr                { Binop($1, And,      $3) }
    | expr OR expr                 { Binop($1, Or,       $3) }
    | ID ASSIGN expr               { Assign($1, $3) }
    | expr COPY expr                 { Copy($1,   $3) }
    | expr MOVE expr                 { Move($1,  $3) }
    | ID LPAREN actuals_opt RPAREN { Call($1,   $3) }
    | ID pathattributes            { Pathattr($1, $2) }
    | ID ADD LPAREN list_expr RPAREN { ListAppend($1, $4) }
    | ID REMOVE LPAREN list_expr RPAREN { ListRemove($1, $4) }

pathattributes:
    | PATHNAME                     { Pathname }
    | PATHCREATED                  { Pathcreated }
    | PATHKIND                     { Pathkind }
    | PATHEXT                      { Pathext }

list_items:
    { Noitem }
    |  expr                         { Item($1) }
    | expr COMMA list_items        { Seq($1, Comma, $3) }


actuals_opt:
    /* nothing */   { [] }
    | actuals_list  { List.rev $1 }

actuals_list:
    expr                      { [$1] }
    | actuals_list COMMA expr { $3 :: $1 }
```

1.3 AST

```
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq | And | Or

type sep = Comma

type data_type = PathType | StrType | IntType | BoolType | VoidType | ListType

type pathattr_type = Pathname | Pathcreated | Pathkind | Pathext

type list_expr =
    ListId of string
  | ListItemInt of int
  | ListItemStr of string

type items =
    Item of expr
  | Seq of expr * sep * items
  | Noitem
and expr =
    LitInt of int
  | LitStr of string
  | Id of string
  | Binop of expr * op * expr
  | Assign of string * expr
  | Call of string * expr list
  | Copy of expr * expr
  | Move of expr * expr
  | List of items
  | ListAppend of string * list_expr
  | ListRemove of string * list_expr
  | Pathattr of string * pathattr_type
  | Noexpr


type for_expr =
    Forid of string

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of for_expr * for_expr * stmt
  (* | For of expr * expr * stmt*)
  | While of expr * stmt
  | Print of expr
  | Ifin of list_expr * list_expr * stmt * stmt

type var_decl = {
  vtype : data_type;
  vname : string;
  vexpr : expr;
}

type func_decl = {
    return : data_type;
    fname : string;
    formals : var_decl list;
    fnlocals : var_decl list;
    body : stmt list;
  }


type program = var_decl list * func_decl list
```

1.4 SAST

```
type op_t = Add | Sub | Mult | Div | Equal | Neq | Less | Leq
          | Greater | Geq | And | Or | StrEqual | StrNeq | StrAdd

type sep_t = Comma

type data_type_t = PathType | StrType | IntType | BoolType | VoidType | ListType

type pathattr_type_t = Pathname | Pathcreated | Pathkind | Pathext

type list_expr_t =
    ListId of string * string
  | ListItemInt of int
  | ListItemStr of string

type items_t =
    Item of expr_t
  | Seq of expr_t * sep_t * items_t
  | Noitem
and expr_t =
    LitInt of int
  | LitStr of string
  | Id of string
  | Binop of expr_t * op_t * expr_t
  | Assign of string * expr_t
  | Call of string * expr_t list
  | Copy of expr_t * expr_t
  | Move of expr_t * expr_t
  | List of items_t
  | ListAppend of string * list_expr_t
  | ListRemove of string * list_expr_t
  | Pathattr of string * pathattr_type_t
  | Noexpr

type for_expr_t =
    Forid of string

type stmt_t =
    Block of stmt_t list
  | Expr of expr_t
  | Return of expr_t
  | If of expr_t * stmt_t * stmt_t
  (*| For of expr_t * expr_t * stmt_t *)
  | For of for_expr_t * for_expr_t * stmt_t
  | While of expr_t * stmt_t
  | Print of expr_t * string
  | Ifin of list_expr_t * list_expr_t * stmt_t * stmt_t

type var_decl_t = {
  vtype : data_type_t;
  vname : string;
  vexpr : expr_t;
}

type func_decl_t = {
    return : data_type_t;
    fname : string;
    formals : var_decl_t list;
    fnlocals : var_decl_t list;
    body : stmt_t list;
  }

type program_t = var_decl_t list * func_decl_t list
```

1.5 Type Check

```
open Ast
open Symboltable

module StringMap = Map.Make(String)

let string_of_vtype = function
  VoidType -> "void"
  | IntType -> "int"
  | StrType -> "string"
  | BoolType -> "bool"
  | PathType -> "path"
  | ListType -> "list"

let get_sast_type = function
  Ast.PathType -> Sast.PathType
  | Ast.StrType -> Sast.StrType
  | Ast.IntType -> Sast.IntType
  | Ast.BoolType -> Sast.BoolType
  | Ast.VoidType -> Sast.VoidType
  | Ast.ListType -> Sast.ListType

let get_sast_pathattrtype = function
  Ast.Pathname -> Sast.Pathname, "string"
  | Ast.Pathcreated -> Sast.Pathcreated, "int"
  | Ast.Pathkind -> Sast.Pathkind, "int"
  | Ast.Pathext -> Sast.Pathext, "string"

let get_vtype env id =
  (* find_variable method is from the symbol table *)
  let t = find_variable id env in
  if t = "" then raise (Failure ("undefined variable: " ^ id)) else t

let get_expr_type t1 t2 =
  if t1 = "void" || t2 = "void" then raise (Failure ("cannot use void type inside expression")) else
  if t1 = "string" || t2 = "string" then "string" else
  if t1 = "int" && t2 = "int" then "int" else
  if t1 = "bool" && t2 = "bool" then "bool" else
  if t1 = "int" && t2 = "bool" then "int" else
  if t1 = "bool" && t2 = "int" then "int" else
  raise (Failure ("type error"))
```

```
let check_listexpr env = function
  | Ast.ListId(id) ->
    Sast.ListId(id, get_vtype env id), get_vtype env id
  | Ast.ListItemInt(i) -> Sast.ListItemInt(i), "int"
  | Ast.ListItemStr(s) -> Sast.ListItemStr(s), "string"


let match_oper e1 op e2 =
  let expr_t = get_expr_type (snd e1) (snd e2) in
  (match op with
     Ast.Add -> if expr_t = "int" then (Sast.Binop(fst e1, Sast.Add, fst e2), "int") else
          if expr_t = "string" then (Sast.Binop(fst e1, Sast.StrAdd, fst e2), "string") else
          raise (Failure ("type error"))
   | Ast.Sub -> if expr_t = "int" then (Sast.Binop(fst e1, Sast.Sub, fst e2), "int") else
      raise (Failure ("type error"))
   | Ast.Mult -> if expr_t = "int" then (Sast.Binop(fst e1, Sast.Mult, fst e2), "int") else
       raise (Failure ("type error"))
   | Ast.Div -> if expr_t = "int" then (Sast.Binop(fst e1, Sast.Div, fst e2), "int") else
      raise (Failure ("type error"))
      (* equal and not equal have special case for string comparison
          we may need to add SAST and Eqs and Neqs *)
   | Ast.Equal -> if expr_t = "int" then (Sast.Binop(fst e1, Sast.Equal, fst e2), "bool") else
          if expr_t = "string" then (Sast.Binop(fst e1, Sast.StrEqual, fst e2), "bool") else
                  raise (Failure ("type error in == "))
   | Ast.Neq -> if expr_t = "int" then (Sast.Binop(fst e1, Sast.Neq, fst e2), "bool") else
          if expr_t = "string" then (Sast.Binop(fst e1, Sast.StrNeq, fst e2), "bool") else
                  raise (Failure ("type error"))
   | Ast.Less ->if expr_t = "int" then (Sast.Binop(fst e1, Sast.Less, fst e2), "bool") else
                  raise (Failure ("type error"))
   | Ast.Leq ->if expr_t = "int" then (Sast.Binop(fst e1, Sast.Leq, fst e2), "bool") else
                  raise (Failure ("type error"))
   | Ast.Greater ->if expr_t = "int" then (Sast.Binop(fst e1, Sast.Greater, fst e2), "bool") else
                  raise (Failure ("type error"))
   | Ast.Geq ->if expr_t = "int" then (Sast.Binop(fst e1, Sast.Geq, fst e2), "bool") else
                  raise (Failure ("type error"))
   | Ast.And ->if expr_t = "bool" then (Sast.Binop(fst e1, Sast.And, fst e2), "bool") else
           raise (Failure ("type error in and"))
   | Ast.Or ->if expr_t = "bool" then (Sast.Binop(fst e1, Sast.Or, fst e2), "bool") else
           raise (Failure ("type error in or"))
  )
```

```
let rec check_expr env = function
  Ast.LitInt(i) -> Sast.LitInt(i), "int"
  | Ast.LitStr(s) -> Sast.LitStr(s), "string"

  | Ast.Id(id) ->
    Sast.Id(id), (get_vtype env id)

  | Ast.Binop(e1, op, e2) ->
    match_oper (check_expr env e1) op (check_expr env e2)

  | Ast.Assign(id, e) ->
    let t = get_vtype env id in
    Sast.Assign(id, (get_expr_with_type env e t)), "void"

  | Ast.Call(func, el) ->
    let args = find_function func env in  (* return & arguments type list from definition *)
    ( match args with
      [] -> raise (Failure ("undefined function " ^ func))
      | hd::tl -> let new_list = try List.fold_left2 check_func_arg [] (List.map (check_expr env) el) tl
                with Invalid_argument "arg" -> raise(Failure("unmatched argument list"))
           in Sast.Call(func, List.rev new_list ), hd )

    | Ast.Move(e1, e2) ->
    let e_t1 = check_expr env e1 in
    let e_t2 = check_expr env e2 in
    if snd e_t1 = "path" && snd e_t2 = "path"
      then Sast.Move(fst e_t1, fst e_t2), "void"
    else
      raise(Failure("cannot use path function on non-path variables"))
    | Ast.Copy(e1, e2) ->
    let e_t1 = check_expr env e1 in
                let e_t2 = check_expr env e2 in
                if snd e_t1 = "path" && snd e_t2 = "path"
                        then Sast.Copy(fst e_t1, fst e_t2), "void"
                else
                        raise(Failure("cannot use path function on non-path variables"))
  | Ast.List(items) -> Sast.List(check_list_items env items), "list"
  | Ast.ListAppend(id, item) -> let t1 = get_vtype env id in
                  let t2 = check_listexpr env item in
                if not (t1 = "list")
                  then raise(Failure("Can append only to id of type list."))
                else if ((snd t2) = "list")
                  then raise(Failure("Cannot append list to list."))
                else
                  Sast.ListAppend( id, (fst t2)), "void"
    | Ast.ListRemove(id, item) -> let t1 = get_vtype env id in
                  let t2 = check_listexpr env item in
                  if not (t1 = "list")
                    then raise(Failure("Can call remove only on type list."))
                  else if ((snd t2) = "list")
                    then raise(Failure("Cannot remove a list from list."))
                  else
                    Sast.ListRemove(id, (fst t2)), "void"
    | Ast.Pathattr(id, e) ->
      if not ((get_vtype env id) = "path")
        then raise(Failure("cannot use path attributes on non-path variable " ^ id))
      else
        Sast.Pathattr(id, fst (get_sast_pathattrtype e)), snd (get_sast_pathattrtype e)
    | Ast.Noexpr -> Sast.Noexpr, "void"

  and check_list_items env = function
      Ast.Item(e) ->let i,t = check_expr env e in
              Sast.Item(i)
    | Ast.Seq(e1, sep, e2) -> Sast.Seq(fst (check_expr env e1), Sast.Comma, (check_list_items env e2))
    | Ast.Noitem -> Sast.Noitem

  and get_expr_with_type env expr t =
    let e = check_expr env expr in
    if ((snd e) = "string" && t = "path") then (fst e)
    else if ((snd e) = "int" && t = "bool") then (fst e)
    else if not((snd e) = t) then raise (Failure ("type error")) else (fst e)

  let check_forexpr env = function
    Ast.Forid(id) -> Sast.Forid(id), get_vtype env id
```

```
let rec check_stmt env func = function
    Ast.Block(stmt_list) -> (Sast.Block(check_stmt_list env func stmt_list)), env
  | Ast.Expr(expr) -> (Sast.Expr(fst (check_expr env expr))), env
  | Ast.Return(expr) -> let e = check_expr env expr in
            if not(snd e = string_of_vtype func.return) then raise (Failure ("The return type doesn't match!"))
            else (Sast.Return(fst e)), env
  | Ast.If(expr, stmt1, stmt2) -> let e = check_expr env expr in
               if not(snd e = "bool") then raise (Failure ("The type of the condition in If statement must be boolean!"))
               else (Sast.If(fst e, fst (check_stmt env func stmt1), fst (check_stmt env func stmt2))), env
  | Ast.Ifin(lexpr1, lexpr2, stmt1, stmt2) -> let e1 = check_listexpr env lexpr1 in
              if (snd e1 = "list") then raise (Failure ("Cannot have list in list!"))
              else let e2 = check_listexpr env lexpr2 in
              if not(snd e2 = "list") then raise (Failure ("\'in\' operator works with list type expression only!"))
            else (Sast.Ifin(fst e1, fst e2, fst (check_stmt env func stmt1), fst (check_stmt env func stmt2))), env
  | Ast.While(expr, stmt) -> let e = check_expr env expr in
               if not (snd e = "bool") then raise (Failure ("The type of the condition in While statement must be boolean!"))
               else (Sast.While(fst e, fst (check_stmt env func stmt))), env        (* while() {} *)
   | Ast.For(expr1, expr2, stmt) -> let e1 = check_forexpr env expr1 in let e2 = check_forexpr env expr2 in
               if not (snd e1 = "path" && snd e2 = "path" ) then raise
                                    (Failure("The type of the expression in a For statement must be path"))
               else (Sast.For(fst e1, fst e2, fst (check_stmt env func stmt))), env
  | Ast.Print(expr) -> let (expr, expr_type) = check_expr env expr in
              (Sast.Print(expr , expr_type)), env

let rec check_expr env = function
  Ast.LitInt(i) -> Sast.LitInt(i), "int"
  | Ast.LitStr(s) -> Sast.LitStr(s), "string"

  | Ast.Id(id) ->
    Sast.Id(id), (get_vtype env id)

  | Ast.Binop(e1, op, e2) ->
    match_oper (check_expr env e1) op (check_expr env e2)

  | Ast.Assign(id, e) ->
    let t = get_vtype env id in
    Sast.Assign(id, (get_expr_with_type env e t)), "void"

  | Ast.Call(func, el) ->
    let args = find_function func env in  (* return & arguments type list from definition *)
    ( match args with
      [] -> raise (Failure ("undefined function " ^ func))
      | hd::tl -> let new_list = try List.fold_left2 check_func_arg [] (List.map (check_expr env) el) tl
               with Invalid_argument "arg" -> raise(Failure("unmatched argument list"))
            in Sast.Call(func, List.rev new_list ), hd )
    (* Need to add type checking for Move and Copy *)
    | Ast.Move(e1, e2) ->
    let e_t1 = check_expr env e1 in
    let e_t2 = check_expr env e2 in
    if snd e_t1 = "path" && snd e_t2 = "path"
      then Sast.Move(fst e_t1, fst e_t2), "void"
    else
      raise(Failure("cannot use path function on non-path variables"))
    | Ast.Copy(e1, e2) ->
    let e_t1 = check_expr env e1 in
               let e_t2 = check_expr env e2 in
               if snd e_t1 = "path" && snd e_t2 = "path"
                       then Sast.Copy(fst e_t1, fst e_t2), "void"
               else
                       raise(Failure("cannot use path function on non-path variables"))
```

```
    | Ast.List(items) -> Sast.List(check_list_items env items), "list"
    | Ast.ListAppend(id, item) -> let t1 = get_vtype env id in
                    let t2 = check_listexpr env item in
                  if not (t1 = "list")
                    then raise(Failure("Can append only to id of type list."))
                  else if ((snd t2) = "list")
                    then raise(Failure("Cannot append list to list."))
                  else
                    Sast.ListAppend( id, (fst t2)), "void"
    | Ast.ListRemove(id, item) -> let t1 = get_vtype env id in
                    let t2 = check_listexpr env item in
                  if not (t1 = "list")
                    then raise(Failure("Can call remove only on type list."))
                  else if ((snd t2) = "list")
                    then raise(Failure("Cannot remove a list from list."))
                  else
                    Sast.ListRemove(id, (fst t2)), "void"
    | Ast.Pathattr(id, e) ->
      if not ((get_vtype env id) = "path")
        then raise(Failure("cannot use path attributes on non-path variable " ^ id))
      else
      (* return type is string assuming path attributes will be treated that way *)
        Sast.Pathattr(id, fst (get_sast_pathattrtype e)), snd (get_sast_pathattrtype e)
    | Ast.Noexpr -> Sast.Noexpr, "void"

  and check_list_items env = function
      Ast.Item(e) ->let i,t = check_expr env e in
              Sast.Item(i)
    | Ast.Seq(e1, sep, e2) -> Sast.Seq(fst (check_expr env e1), Sast.Comma, (check_list_items env e2))
    | Ast.Noitem -> Sast.Noitem

  and get_expr_with_type env expr t =
    let e = check_expr env expr in
    (* added special case for the path variable *)
    if ((snd e) = "string" && t = "path") then (fst e)
    else if ((snd e) = "int" && t = "bool") then (fst e)
    else if not((snd e) = t) then raise (Failure ("type error")) else (fst e)


  let check_forexpr env = function
    Ast.Forid(id) -> Sast.Forid(id), get_vtype env id
```

```
let rec check_stmt env func = function
    Ast.Block(stmt_list) -> (Sast.Block(check_stmt_list env func stmt_list)), env
  | Ast.Expr(expr) -> (Sast.Expr(fst (check_expr env expr))), env
  | Ast.Return(expr) -> let e = check_expr env expr in
            if not(snd e = string_of_vtype func.return) then raise (Failure ("The return type doesn't match!"))
            else (Sast.Return(fst e)), env
  | Ast.If(expr, stmt1, stmt2) -> let e = check_expr env expr in
                if not(snd e = "bool") then raise (Failure ("The type of the condition in If statement must be boolean!"))
                else (Sast.If(fst e, fst (check_stmt env func stmt1), fst (check_stmt env func stmt2))), env  (* if() {} else{} *)
  | Ast.Ifin(lexpr1, lexpr2, stmt1, stmt2) -> let e1 = check_listexpr env lexpr1 in
                if (snd e1 = "list") then raise (Failure ("Cannot have list in list!"))
              else let e2 = check_listexpr env lexpr2 in
              if not(snd e2 = "list") then raise (Failure ("\'in\' operator works with list type expression only!"))
            else (Sast.Ifin(fst e1, fst e2, fst (check_stmt env func stmt1), fst (check_stmt env func stmt2))), env
  | Ast.While(expr, stmt) -> let e = check_expr env expr in
               if not (snd e = "bool") then raise (Failure ("The type of the condition in While statement must be boolean!"))
               else (Sast.While(fst e, fst (check_stmt env func stmt))), env        (* while() {} *)
   | Ast.For(expr1, expr2, stmt) -> let e1 = check_forexpr env expr1 in let e2 = check_forexpr env expr2 in
               if not (snd e1 = "path" && snd e2 = "path" ) then raise (Failure("The type of the expression in a For statement must be path"))
               else (Sast.For(fst e1, fst e2, fst (check_stmt env func stmt))), env
  | Ast.Print(expr) -> let (expr, expr_type) = check_expr env expr in
              (Sast.Print(expr , expr_type)), env

and check_stmt_list env func = function
    [] -> []
  | hd::tl -> let s,e = (check_stmt env func hd) in s::(check_stmt_list e func tl)

let convert_to_sast_type x env =
  let t = get_vtype env x.vname in
    let s_expr =
    if not (x.vexpr = Ast.Noexpr) then
      get_expr_with_type env x.vexpr t
    else Sast.Noexpr
    in
  {
    Sast.vtype = get_sast_type x.vtype;
    Sast.vname = x.vname;
    Sast.vexpr = s_expr;
  }

let check_formal env formal =
  let ret = add_local formal.vname formal.vtype env in
  if (string_of_vtype formal.vtype) = "void" then raise (Failure("cannot use void as variable type")) else
  if StringMap.is_empty ret then raise (Failure ("local variable " ^ formal.vname ^ " is already defined"))
  else let env = {locals = ret; globals = env.globals; functions = env.functions } in
  convert_to_sast_type formal env, env

let rec check_formals env formals =
  match formals with
    [] -> []
  | hd::tl -> let f, e = (check_formal env hd) in (f, e)::(check_formals e tl)

let check_local env local =
  let ret = add_local local.vname local.vtype env in
  if (string_of_vtype local.vtype) = "void" then raise (Failure("cannot use void as variable type")) else
  if StringMap.is_empty ret then raise (Failure ("local variable " ^ local.vname ^ " is already defined"))
  else let env = {locals = ret; globals = env.globals; functions = env.functions } in
  convert_to_sast_type local env, env
```

```
let rec check_locals env locals =
  match locals with
    [] -> []
  | hd::tl -> let l, e = (check_local env hd) in (l, e)::(check_locals e tl)

let check_function env func =
  match List.hd (List.rev func.body) with
  Return(_) ->
      let env = {locals = StringMap.empty; globals = env.globals; functions = env.functions } in
      (*  ret is new env *)
    let ret = add_function func.fname func.return func.formals env in
    if StringMap.is_empty ret then raise (Failure ("function " ^ func.fname ^ " is already defined"))
    else let env = {locals = env.locals; globals = env.globals; functions = ret } in
    let f = check_formals env func.formals in
    let formals = List.map (fun formal -> fst formal) f in

    (* get the final env from the last formal *)
    let l, env =
    (match f with
        [] -> let l = check_locals env func.fnlocals in
          l, env
      | _ ->  let env = snd (List.hd (List.rev f)) in
          let l = check_locals env func.fnlocals in
          l, env
    ) in
    let fnlocals = List.map (fun fnlocal -> fst fnlocal) l in
     (match l with
              [] -> let body = check_stmt_list env func func.body in
                  { Sast.return = get_sast_type func.return;
                    Sast.fname = func.fname;
                    Sast.formals = formals;
                    Sast.fnlocals = fnlocals;
                    Sast.body = body
                  }, env
              | _ -> let e = snd (List.hd (List.rev l)) in
                     let body = check_stmt_list e func func.body in
                    { Sast.return = get_sast_type func.return;
                      Sast.fname = func.fname;
                      Sast.formals = formals;
                      Sast.fnlocals = fnlocals;
                      Sast.body = body
                    }, e
          )
  | _ -> raise (Failure ("The last statement must be return statement"))

let rec check_functions env funcs =
  match funcs with
    [] -> []
  | hd::tl -> let f, e = (check_function env hd) in f::(check_functions e tl)

let check_global env global =
  if (string_of_vtype global.vtype) = "void" then raise (Failure("cannot use void as variable type"))
  else let ret = add_global global.vname global.vtype env in
  if StringMap.is_empty ret then raise (Failure ("global variable " ^ global.vname ^ " is already defined"))
  (* update the env with globals from ret *)
  else let env = {locals = env.locals; globals = ret; functions = env.functions } in
  convert_to_sast_type global env, env

let rec check_globals env globals =
  match globals with
    [] -> []
  | hd::tl -> let g, e = (check_global env hd) in (g, e)::(check_globals e tl)

let check_program (globals, funcs) =
  let env = { locals = StringMap.empty;
        globals = StringMap.empty;
        functions = StringMap.empty }
  in
  let g = check_globals env globals in
  let globals = List.map (fun global -> fst global) g in
  match g with
   [] -> (globals, (check_functions env (List.rev funcs)))
  | _ -> let e = snd (List.hd (List.rev g)) in (globals, (check_functions e (List.rev funcs)))
```

1.6 Symbol Table

```
1  open Ast
2
3  module StringMap = Map.Make(String)
4
5  type env = {
6      locals:         string StringMap.t;
7    globals:        string StringMap.t;
8    functions:      string list StringMap.t;
9  }
10
11 let string_of_vtype = function
12   VoidType -> "void"
13   | IntType -> "int"
14   | StrType -> "string"
15   | BoolType -> "bool"
16   | PathType -> "path"
17   | ListType -> "list"
18
19 let find_variable name env =
20   try StringMap.find name env.locals
21   with Not_found -> try StringMap.find name env.globals
22   with Not_found -> ""
23   (*raise (Failure ("undefined variable " ^ name)) *)
24
25 let find_function name env =
26   try StringMap.find name env.functions
27   with Not_found -> []
28   (*raise (Failure ("undefined function " ^ name)) *)
29
30 let add_local name v_type env =
31   if StringMap.mem name env.locals then StringMap.empty
32   else StringMap.add name (string_of_vtype v_type) env.locals
33
34 let add_global name v_type env =
35   if StringMap.mem name env.globals then StringMap.empty
36   else StringMap.add name (string_of_vtype v_type) env.globals
37
38 (* from the ast *)
39 let get_arg_type = function
40   v -> string_of_vtype v.vtype
41
42 let add_function name return_type formals env =
43   if StringMap.mem name env.functions then StringMap.empty
44   else let f = List.map get_arg_type formals in
45   StringMap.add name (string_of_vtype (return_type)::f) env.functions
```

2 Shell Scripts

2.1 Test All Script

```
#!/bin/sh

if [ ! -f "c/libraries/liblist.a" ] || [ ! -f "c/libraries/libpath.a" ] ; then
    cd c/libraries
    make >> lib_msgs.txt
    cd ../..
fi

if [ ! -f "preprocessor/./preprocessor" ]; then
    cd preprocessor
    make >> preproc_msgs.txt
    cd ..
fi

if [ ! -f "./fdl" ]; then
    make >> compiler_msgs.txt
fi

FDL="./fdl"
PRE="preprocessor/./preprocessor"

Compare() {
  difference=$(diff -b $1 $2)
  echo $difference
  if [ "$difference" != "" ]; then
    echo $difference > $3
  fi
}

function compile() {
  basename=`echo $1 | sed 's/.*\\///
                           s/.fdl//'`
  reffile=`echo $1 | sed 's/.fdl$//'`
    prepfile=${reffile}'.fdlp'
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/"

  testoutput=`echo ${basedir}test_outputs/$basename.c.out`

    echo "Preprocessing '$1'..."
    $PRE $1 $prepfile && echo "Preprocessor for $1 succeeded"

  echo "Compiling $prepfile ..."
    $FDL $prepfile > "${reffile}.c" && echo "Ocaml to C of $1 succeeded"

    if [ -f "${reffile}.c" ]; then
      gcc -Ic/libraries -Lc/libraries -llist -lpath "${reffile}.c" -o "${reffile}" && echo "COMPILATION of ${reffile}.c succeeded"
    else
      echo "Ocaml to C of $1 failed"
      return
    fi

    rm -rf $prepfile

    if [ -f "${reffile}" ]; then
      eval ${reffile} > ${reffile}.generated.out
      Compare ${testoutput} ${reffile}.generated.out ${reffile}.c.diff
        rm -rf ${reffile}.generated.out
        rm -rf ${reffile}.c
        rm -rf ${reffile}
    else
      echo "C to binary of ${reffile}.c failed"
    fi
}

files=sample_program/*.fdl

for file in $files
do
  compile $file
done
```

2.2 Run FDL Script

```
#!/bin/sh

if [ ! -f "c/libraries/liblist.a" ] || [ ! -f "c/libraries/libpath.a" ] ; then
    cd c/libraries
    make >> lib_msgs.txt
    cd ../..
fi

if [ ! -f "preprocessor/./preprocessor" ]; then
    cd preprocessor
    make >> preproc_msgs.txt
    cd ..
fi

if [ ! -f "./fdl" ]; then
    make >> compiler_msgs.txt
fi

# fdl exectutable
FDL="./fdl"
# preprocessor executable
PRE="./preprocessor/preprocessor"

function compileAndRun() {
  basename=`echo $1 | sed 's/.*\\///
                            s/.fdl//'`
  reffile=`echo $1 | sed 's/.fdl$//'`
    prepfile=$reffile'.fdlp'
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/"


  $PRE $1 $prepfile

  if [ ! -f $prepfile ]; then
    echo "$prepfile does not exist"
        return
  fi

    $FDL $prepfile > "${reffile}.c"

    if [ -f "${reffile}.c" ]; then
      gcc -Ic/libraries -Lc/libraries -llist -lpath -w -o "${reffile}" "${reffile}.c"
    else
      echo "Ocaml to C of $1 failed"
      return
    fi

    if [ -f "${reffile}" ]; then
        eval ${reffile}
        rm -rf ${reffile}.fdlp
        rm -rf ${reffile}.c
        rm -rf ${reffile}
    else
        echo "C to binary of ${reffile}.c failed"
    fi
}

if [ -f $1 ]; then
  compileAndRun $1
else
  echo "$1 doesnt exist"
fi
```

2.3 Clean All Script

```
#!/bin/sh

if [ -f "c/libraries/liblist.a" ] || [ -f "c/libraries/libpath.a" ] ; then
    cd c/libraries
    make clean
    cd ../..
fi

if [ -f "preprocessor/./preprocessor" ]; then
    cd preprocessor
    make clean
    cd ..
fi

if [ -f "./fdl" ]; then
    make clean
fi
```

3 Preprocessor

3.1 Makefile

```
CC  = gcc
CXX = g++

INCLUDES =
CFLAGS   = -g -Wall $(INCLUDES)
CXXFLAGS = -g -Wall $(INCLUDES)

LDFLAGS =
LDLIBS =

.PHONY: default
default: preprocessor

# header dependency
preprocessor: preprocessor.o

.PHONY: clean
clean:
  rm -f *.o *.txt *~ a.out core preprocessor

.PHONY: all
all: clean default
```

3.2 Preprocessor

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <assert.h>
#include <ctype.h>  /* For isspace(). */
#include <stddef.h>  /* For size_t. */

#define MAX_BUFFER 4096

static void die(const char *message)
{
    perror(message);
    exit(1);
}

const char *getFileExtension(const char *fileName) {
    const char *dot = strrchr(fileName, '.');
    if(!dot || dot == fileName) return "";
    return dot + 1;
}

void remove_whitespace(char *str) {
    char *p;
    size_t len = strlen(str);

    for(p = str; *p; p ++, len --) {
        while(isspace(*p)) memmove(p, p+1, len--);
    }
}

int is_empty(const char *s) {
  while (*s != '\0') {
    if (!isspace(*s))
      return 0;
    s++;
  }
  return 1;
}


int main(int argc, char const *argv[])
{
    if (argc != 3) {
        fprintf(stderr, "%s\n", "usage: ./preprocessor <fdl file> <fdlp file>");
        exit(1);
    }
    char *fileName = (char *) argv[1];
    char *outputFileName = (char *) argv[2];

    if (strcmp("fdl", getFileExtension(fileName)) != 0)
    {
        die("file extension must be fdl");
    }
    if (strcmp("fdlp", getFileExtension(outputFileName)) != 0)
    {
        die("output file extension must be fdlp");
    }
    FILE *input;
    if ((input = fopen(fileName, "r")) == NULL) {
        die("fpen() failed");
    }
    FILE *output;
    if ((output = fopen(outputFileName, "w")) == NULL) {
        die("fpen() failed");
    }

    char buffer[MAX_BUFFER];
```

```
    while (fgets(buffer, sizeof(buffer), input) != NULL) {
        size_t len = strlen(buffer) - 1;
        if (buffer[len] == '\n') {
            buffer[len] = '\0';
        }
        if (strstr(buffer, "*/") != NULL) {
            fprintf(output, "%s\n", buffer);
        }
        else if (strstr(buffer, "/*") != NULL) {
            fprintf(output, "%s\n", buffer);
        }
        else if (strstr(buffer, "def ") != NULL) {
            fprintf(output, "%s {\n", buffer);
        }
        else if (strstr(buffer, "int ") != NULL) {
            fprintf(output, "%s;\n", buffer);
        }
        else if (strstr(buffer, "path ") != NULL) {
            fprintf(output, "%s;\n", buffer);
        }
        else if (strstr(buffer, "dict ") != NULL) {
            fprintf(output, "%s;\n", buffer);
        }
        else if (strstr(buffer, "list ") != NULL) {
            fprintf(output, "%s;\n", buffer);
        }
        else if (strstr(buffer, "string ") != NULL) {
            fprintf(output, "%s;\n", buffer);
        }
        else if (strstr(buffer, "bool ") != NULL) {
            fprintf(output, "%s;\n", buffer);
        }
        else if (strstr(buffer, "for ") != NULL) {
            fprintf(output, "%s {\n", buffer);
        }
        else if ((strstr(buffer, "if (") != NULL || strstr(buffer, "if(") != NULL)
                                            && (strstr(buffer, "then") != NULL)) {
            fprintf(output, "%s {\n", buffer);
        }
        else if ((strstr(buffer, "if (") != NULL || strstr(buffer, "if(") != NULL)
                                            && (strstr(buffer, "then") == NULL)) {
            fprintf(output, "%s\n", buffer);
        }
        else if (strstr(buffer, "then") != NULL) {
            fprintf(output, "%s {\n", buffer);
        }
```

```
115    else if (strstr(buffer, "else") != NULL) {
116                int i;
117                int counter = 0;
118                for (i = 0; i < strlen(buffer); ++i)
119                {
120                    if (buffer[i] == ' ') {
121                        fprintf(output, "%c", buffer[i]);
122                        counter++;
123                    }
124                }
125                fprintf(output, "} %s {\n", buffer + counter);
126            }
127            else if (strstr(buffer, "while (") != NULL || strstr(buffer, "while(") != NULL) {
128                fprintf(output, "%s {\n", buffer);
129            }
130            else if (strstr(buffer, "end") != NULL) {
131                int i;
132                for (i = 0; i < strlen(buffer); i++){
133                    if (buffer[i] == 'e') {
134                        buffer[i] = '}';
135                    } else if (buffer[i] == 'n') {
136                        buffer[i] = '\n';
137                    } else if (buffer[i] == 'd') {
138                        buffer[i] = '\0';
139                    } else {
140
141                    }
142                }
143                fprintf(output, "%s", buffer);
144            }
145            else {
146                if (is_empty(buffer)) {
147                    remove_whitespace(buffer);
148                    fprintf(output, "\n");
149                } else {
150                    fprintf(output, "%s;\n", buffer);
151                }
152            }
153        }
154        fclose(input);
155        fclose(output);
156        return 0;
157    }
```

4 Libraries in C

4.1 Makefile

```
CC  = gcc
CXX = g++

INCLUDES = -I libraries/

CFLAGS   = -g -Wall $(INCLUDES)
CXXFLAGS = -g -Wall $(INCLUDES)

LDFLAGS = -g -L libraries/

LDLIBS = -llist -lpath

stat_calls: stat_calls.o

stat_calls.o: stat_calls.c

.PHONY: clean
clean:
    rm -f *.o *.txt a.out core stat_calls

.PHONY: all
all: clean stat_calls
```

4.2 Lists

4.2.1 List Header

```
#ifndef _LIST_H_
#define _LIST_H_

enum fdl_type { fdl_str, fdl_path, fdl_int, fdl_bool };

struct Node {
  enum fdl_type type;
    union {
        int int_item;
        int bool_item;
        char *string_item;
        char *path_item;
    };
    struct Node *next;
};

struct List {
    struct Node *head;
};

struct Node *createIntNode(int data, enum fdl_type type);
struct Node *createStrNode(char *data, enum fdl_type type);

static inline void initList(struct List *list)
{
    list->head = 0;
}

static inline int isEmptyList(struct List *list)
{
    return (list->head == 0);
}

void addFront(struct List *list, struct Node *node);
void traverseList(struct List *list, void (*f)(struct Node *));
void printNode(struct Node *node);
int findNode(struct List *list, struct Node *node1);
void removeNode(struct List *list, struct Node *node1);
struct Node popFront(struct List *list);
void removeAllNodes(struct List *list);
void addAfter(struct List *list, struct Node *prevNode, struct Node *newNode);
void reverseList(struct List *list);
void addBack(struct List *list, struct Node *newNode);
void loadDirectoryToList(char *path, struct List *subPath);

#endif
```

4.2.2 List Implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "list.h"
#include "dirent.h"

void loadDirectoryToList(char *path, struct List *subPath){
    char *buffer;
    DIR *dir;
    struct dirent *ent;
    int len;
    if ((dir = opendir (path)) != NULL) {
        /* print all the files and directories within directory */
        while ((ent = readdir (dir)) != NULL) {
            len = strlen(path) + strlen(ent->d_name) + 2;
            buffer = (char *)malloc(sizeof(char)*len);
            //printf("%s\n",ent->d_name);
            strcpy(buffer, path);
            strcat(buffer, "/");
            strcat(buffer, ent->d_name);
            struct Node * node = createStrNode(buffer, fdl_path);
            addBack(subPath, node);
            //buffer = "\0";
        }
        closedir (dir);
    } else {
        /* could not open directory */
        perror ("");
        exit(0);
    }
}

struct Node *createIntNode(int data, enum fdl_type type) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    if(newNode == NULL){
        printf("Could not create new node!\n");
        exit(1);
    }

    newNode->type = type;
    newNode->next = NULL;
    switch(newNode->type){
        case fdl_int: newNode->int_item = data; break;
        case fdl_bool: newNode->bool_item = data; break;
        default: break;
    }
    return newNode;
}

struct Node *createStrNode(char *data, enum fdl_type type) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    if(newNode == NULL){
        printf("Could not create new node!\n");
        exit(1);
    }

    newNode->type = type;
    newNode->next = NULL;
    switch(newNode->type){
        case fdl_str: newNode->string_item = data; break;
        case fdl_path: newNode->path_item = data; break;
        default: break;
    }
    return newNode;
}

void addFront(struct List *list, struct Node *node)
{
    node->next = list->head;
    list->head = node;
}
```

```
72  void traverseList(struct List *list, void (*f)(struct Node *))
73  {
74      struct Node *node = list->head;
75      while (node) {
76      f(node);
77      node = node->next;
78      }
79  }
80
81  void printNode(struct Node *node)
82  {
83      switch(node->type){
84          case fdl_int: printf("%d\n",node->int_item); break;
85          case fdl_bool: if(node->bool_item == 1) printf("True\n");
86                         else printf("False\n"); break;
87          case fdl_str: printf("%s\n",node->string_item); break;
88          case fdl_path: printf("%s\n",node->path_item); break;
89      }
90  }
91
92  int findNode(struct List *list, struct Node *node1) {
93      struct Node *node2 = list->head;
94      while (node2) {
95          if(node1->type == node2->type){
96              switch(node1->type){
97                  case fdl_int: if (node1->int_item == node2->int_item) return 0; else break;
98                  case fdl_str: if (strcmp(node1->string_item, node2->string_item) == 0) return 0; else break;
99                  case fdl_bool: if (node1->bool_item == node2->bool_item) return 0; else break;
100                 case fdl_path: if (strcmp(node1->path_item, node2->path_item) == 0) return 0; else break;
101                 default: return 1;
102             }
103         }
104         node2 = node2->next;
105     }
106     return 1;
107 }
108
109 void removeNode(struct List *list, struct Node *node1) {
110     struct Node *node2 = list->head;
111     int del = 0;
112     struct Node *prev = list->head;
113     while (node2) {
114         if(node1->type == node2->type){
115             switch(node1->type){
116                 case fdl_int: if (node1->int_item == node2->int_item) { del = 1; break; } else break;
117                 case fdl_str: if (strcmp(node1->string_item, node2->string_item) == 0) { del = 1; break; } else break;
118                 case fdl_bool: if (node1->bool_item == node2->bool_item) { del = 1; break; } else break;
119                 case fdl_path: if (strcmp(node1->path_item, node2->path_item) == 0) { del = 1; break; } else break;
120                 default: del = 0;
121             }
122         }
123         if(del == 0){
124             prev = node2;
125             node2 = node2->next;
126         }
127         else break;
128     }
129     if(del == 1){
130         if(node2 == list->head)
131             list->head = node2->next;
132         else
133             prev->next = node2->next;
134         free(node2);
135     }
136     else {
137         printf("List item not found.\n");
138     }
139 }
140
141 struct Node popFront(struct List *list) {
142     struct Node *oldHead = list->head;
143     struct Node node = *oldHead;
144     list->head = oldHead->next;
145     free(oldHead);
146     return node;
147 }
```

```
148  void removeAllNodes(struct List *list)
149  {
150      while (!isEmptyList(list))
151      popFront(list);
152  }
153
154  void addAfter(struct List *list,
155      struct Node *prevNode, struct Node *newNode)
156  {
157      if (prevNode == NULL)
158         addFront(list, newNode);
159
160      newNode->next = prevNode->next;
161      prevNode->next = newNode;
162  }
163
164  void reverseList(struct List *list)
165  {
166      struct Node *prv = NULL;
167      struct Node *cur = list->head;
168      struct Node *nxt;
169
170      while (cur) {
171      nxt = cur->next;
172      cur->next = prv;
173      prv = cur;
174      cur = nxt;
175      }
176
177      list->head = prv;
178  }
179
180  void addBack(struct List *list, struct Node *newNode)
181  {
182      newNode->next = NULL;
183
184      if (list->head == NULL) {
185         list->head = newNode;
186         return;
187      }
188
189      struct Node *end = list->head;
190      while (end->next != NULL)
191         end = end->next;
192
193      end->next = newNode;
194  }
```

4.3 Paths

4.3.1 Path Header

```
#ifndef _PATH_H_
#define _PATH_H_

char* getName(char *path, char *output);
int checkValid(char *path);
int getCreatedAt(char *path);
int getPathType(char *path);
int isDir(char *path);
char* getPathName(char *path);
int copyFile(char* src, char *dest);
int moveFile(char* src, char *dest);
char* getExtension(char *path);
char* stringConcat(char *str1, char *str2);

#endif
```

4.3.2 Path Implementation

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sys/stat.h"
#include "time.h"
#include<libgen.h>

// test function
char* getName(char *path, char *output){
  char *dirc, *basec, *bname, *dname;

  dirc = strdup(path);
  basec = strdup(path);
  dname = dirname(dirc);
  bname = basename(basec);
  //printf("dirname=%s, basename=%s\n", dname, bname);
  strcpy(output, dname);
  return output;
}

int checkValid(char *path){
  /* testing the stat sys call for files and directories */
  struct stat info;
  if (stat(path, &info) != 0)
    return 0;
  else
    // can be valid directory or file
    return S_ISDIR(info.st_mode) ? 1 : S_ISREG(info.st_mode);
}

// returns -1 in case of invalid path
int getCreatedAt(char *path){
  if(checkValid(path)){
    struct stat info;
    stat(path, &info);

    return (int) info.st_birthtime;
  }else
    return -1;
}

// Directory 1, File 0, invalid path -1
int getPathType(char *path){
  if(checkValid(path)){
    struct stat info;
    stat(path, &info);

    return S_ISDIR(info.st_mode);
  }else
    return -1;
}

int isDir(char *path){
  if(checkValid(path)){
    struct stat info;
    stat(path, &info);

    return S_ISDIR(info.st_mode);
  }else
    return -1;
}

// get the last directory or filename
char* getPathName(char* path){
  if(checkValid(path)){
        char *basec = strdup(path);
        char *bname = basename(basec);
        return bname;
    }else
        return NULL;

}
```

```
73  int copyFile(char* src, char *dest){
74    char copycommand[1000];
75    if (checkValid(dest) == 0) {
76      char temp[1000] = "mkdir -p ";
77      strcat(temp, dest);
78      system(temp);
79    }
80    sprintf(copycommand, "/bin/cp %s %s", src, dest);
81    return system(copycommand);
82  }
83
84  int moveFile(char* src, char *dest){
85    char movecommand[1000];
86
87    if (checkValid(dest) == 0) {
88      char temp[1000] = "mkdir -p ";
89      strcat(temp, dest);
90      system(temp);
91    }
92    sprintf(movecommand, "/bin/mv %s %s", src, dest);
93    return system(movecommand);
94  }
95
96  char* getExtension(char *path){
97    char *ptr = rindex(path, '.');
98    return strdup(ptr);
99  }
100
101 char* stringConcat(char *str1, char *str2){
102   char *strdup1 = strdup(str1);
103   char *strdup2 = strdup(str2);
104   strcat(strdup1, strdup2);
105   return strdup1;
106 }
```