

1 O'Caml

1.1 Scanner

```
1 { open Parser }
2
3 let letter = ['a' - 'z' 'A' - 'Z']
4 let digit = ['0' - '9']
5 let quote = ""
6
7 rule token = parse
8   [' ' '\r' '\n' '\t'] { token lexbuf } | "/" * " { comment lexbuf }
9   | '(' { LPAREN } | ')' { RPAREN }
10  | '{' { LBRACE } | '}' { RBRACE } | ',' { COMMA }
11  | '+' { PLUS } | '-' { MINUS }
12  | '*' { TIMES } | '/' { DIVIDE }
13  | '=' { ASSIGN } | ';' { SEMI }
14  | "<<-" { MOVE }
15  | "<-" { COPY }
16  | "==" { EQ } | "!=" { NEQ }
17  | '<' { LT } | "<=" { LEQ }
18  | '>' { GT } | ">=" { GEQ }
19  | '[' { LBRACK } | ']' { RBRACK }
20  | "&&" { AND } | "||" { OR }
21  | '!' { NOT } | ".name" { PATHNAME }
22  | "def" { DEF } | ".created_at" { PATHCREATED }
23  | "int" { INT } | ".kind" { PATHKIND }
24  | "path" { PATH }
25  | "string" { STR } | "list" { LIST }
26  | "if" { IF } | "else" { ELSE }
27  | "then" { THEN } | "print" { PRINT }
28  | "for" { FOR } | "in" { IN }
29  | "do" { DO } | "bool" { BOOL }
30  | "while" { WHILE } | "return" { RETURN }
31  | "void" { VOID } | ".add" { ADD }
32  | "true" { TRUE } | ".remove" { REMOVE }
33  | "false" { FALSE } | ".type" { PATHEXT }
34  | "trash" { TRASH }
35  | eof { EOF } (* do as microC *)
36  | digit+ as lit { LIT_INT(int_of_string lit) }
37  | quote [^']* quote as lit { LIT_STR(lit) }
38  | letter | (letter | digit | '_' ) * as id { ID(id) }
39  | _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }
40
41 and comment = parse
42   "/" * " { token lexbuf }
43   | _ { comment lexbuf }
```

1.2 Parser

```
1  %{ open Ast %}
2
3  %token LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK COMMA TAB SEMI
4  %token PLUS MINUS TIMES DIVIDE ASSIGN MOVE COPY
5  %token EQ NEQ LT LEQ GT GEQ NOT
6  %token AND OR
7  %token RETURN IF THEN ELSE FOR IN WHILE DO
8  %token DEF VOID INT STR LIST PATH BOOL TRASH TRUE FALSE PRINT
9  %token PATHNAME PATHCREATED PATHKIND PATHEXT ADD REMOVE
10 %token <int> LIT_INT
11 %token <string> LIT_STR
12 %token <bool> LIT_BOOL
13 %token <string> ID
14 %token IN
15 %token EOF
16
17 %nonassoc NOELSE
18 %nonassoc ELSE
19
20 %right ASSIGN MOVE COPY NOT
21
22 %left AND OR
23 %left EQ NEQ
24 %left LT GT LEQ GEQ
25 %left IN
26 %left PLUS MINUS
27 %left TIMES DIVIDE
28
29 %start program
30 %type <Ast.program> program
31
32 %%
33
34 program:
35     { [], [] }
36     | program vdecl { ($2 :: fst $1), snd $1 }
37     | program fdecl { fst $1, ($2 :: snd $1) }
38
39 fdecl:
40     DEF return_type ID LPAREN formals_opt RPAREN LBRACE vdecl_opt stmt_list RBRACE
41     {{
42         return = $2;
43         fname = $3;
44         formals = $5;
45         fnlocals = List.rev $8;
46         body = List.rev $9 }}
```

```

47 return_type:
48     VOID      { VoidType }
49     | INT      { IntType }
50     | BOOL     { BoolType }
51     | PATH     { PathType }
52     | STR      { StrType }
53     | LIST     { ListType }
54
55 forms_opt:
56     { [] }
57     | formal_list { List.rev $1 }
58
59 formal_list:
60     formal      { [$1] }
61     | formal_list COMMA formal { $3 :: $1 }
62
63 formal:
64     INT ID      { { vtype = IntType;  vname = $2; vexpr = Noexpr; } }
65     | BOOL ID   { { vtype = BoolType; vname = $2; vexpr = Noexpr; } }
66     | PATH ID   { { vtype = PathType; vname = $2; vexpr = Noexpr; } }
67     | STR ID    { { vtype = StrType;  vname = $2; vexpr = Noexpr; } }
68     | LIST ID   { { vtype = ListType; vname = $2; vexpr = Noexpr; } }
69
70 vdecl_opt:
71     { [] }
72     | vdecl_list { List.rev $1 }
73
74 vdecl_list:
75     vdecl      { [$1] }
76     | vdecl_list vdecl { $2 :: $1 }
77
78 vdecl:
79     vdecl_type ID SEMI { { vtype = $1;  vname = $2; vexpr = Noexpr; } }
80     | vdecl_type ID ASSIGN expr SEMI { { vtype = $1;  vname = $2; vexpr = $4; } }
81
82 vdecl_type:
83     VOID      { VoidType }
84     | INT      { IntType }
85     | BOOL     { BoolType }
86     | STR      { StrType }
87     | PATH     { PathType }
88     | LIST     { ListType }
89
90 stmt_list:
91     { [] }
92     | stmt_list stmt { $2 :: $1 }
93
94 rev_stmt_list:
95     stmt_list { List.rev $1 }
96
97 stmt:
98     expr SEMI { Expr($1) }
99     | RETURN expr_opt SEMI { Return($2) }
100    | IF LPAREN expr RPAREN THEN stmt %prec NOELSE { If($3, $6, Block([])) }
101    | IF LPAREN expr RPAREN THEN stmt ELSE stmt { If($3, $6, $8) }
102    | PRINT expr SEMI { Print($2) }
103    | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
104    | FOR LPAREN for_expr IN for_expr RPAREN stmt { For($3, $5, $7) }
105    | IF list_expr IN list_expr THEN stmt %prec NOELSE { Ifin($2, $4, $6, Block([])) }
106    | IF list_expr IN list_expr THEN stmt ELSE stmt { Ifin($2, $4, $6, $8) }
107    | LBRACE rev_stmt_list RBRACE { Block($2) }
108
109
110 for_expr:
111     ID { Forid($1) }
112
113 list_expr:
114     ID { ListId($1) }
115     | LIT_INT { ListItemInt($1) }
116     | LIT_STR { ListItemStr($1) }
117
118 expr_opt:
119     /* nothing */ { Noexpr }
120     | expr { $1 }

```

```

121 expr:
122   | LIT_INT                { LitInt($1) }
123   | TRUE                   { LitInt(1) }
124   | FALSE                   { LitInt(0) }
125   | LIT_STR                 { LitStr($1) }
126   | LBRACK list_items RBRACK { List($2) }
127   | ID                      { Id($1) }
128   | expr PLUS expr          { Binop($1, Add, $3) }
129   | expr MINUS expr         { Binop($1, Sub, $3) }
130   | expr TIMES expr         { Binop($1, Mult, $3) }
131   | expr DIVIDE expr        { Binop($1, Div, $3) }
132   | expr EQ expr            { Binop($1, Equal, $3) }
133   | expr NEQ expr           { Binop($1, Neq, $3) }
134   | expr LT expr            { Binop($1, Less, $3) }
135   | expr LEQ expr           { Binop($1, Leq, $3) }
136   | expr GT expr            { Binop($1, Greater, $3) }
137   | expr GEQ expr           { Binop($1, Geq, $3) }
138   | expr AND expr           { Binop($1, And, $3) }
139   | expr OR expr            { Binop($1, Or, $3) }
140   | ID ASSIGN expr          { Assign($1, $3) }
141   | expr COPY expr          { Copy($1, $3) }
142   | expr MOVE expr          { Move($1, $3) }
143   | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
144   | ID pathattributes        { Pathattr($1, $2) }
145   | ID ADD LPAREN list_expr RPAREN { ListAppend($1, $4) }
146   | ID REMOVE LPAREN list_expr RPAREN { ListRemove($1, $4) }
147
148 pathattributes:
149   | PATHNAME                { Pathname }
150   | PATHCREATED              { Pathcreated }
151   | PATHKIND                 { Pathkind }
152   | PATHEXT                  { Pathext }
153
154 list_items:
155   { Noitem }
156   | expr                    { Item($1) }
157   | expr COMMA list_items   { Seq($1, Comma, $3) }
158
159
160 actuals_opt:
161   /* nothing */ { [] }
162   | actuals_list { List.rev $1 }
163
164 actuals_list:
165   expr { [$1] }
166   | actuals_list COMMA expr { $3 :: $1 }

```

1.3 AST

```
1  type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq | And | Or
2
3  type sep = Comma
4
5  type data_type = PathType | StrType | IntType | BoolType | VoidType | ListType
6
7  type pathattr_type = Pathname | Pathcreated | Pathkind | Pathext
8
9  type list_expr =
10     ListId of string
11     | ListItemInt of int
12     | ListItemStr of string
13
14  type items =
15     Item of expr
16     | Seq of expr * sep * items
17     | Noitem
18  and expr =
19     LitInt of int
20     | LitStr of string
21     | Id of string
22     | Binop of expr * op * expr
23     | Assign of string * expr
24     | Call of string * expr list
25     | Copy of expr * expr
26     | Move of expr * expr
27     | List of items
28     | ListAppend of string * list_expr
29     | ListRemove of string * list_expr
30     | Pathattr of string * pathattr_type
31     | Noexpr
32
33
34  type for_expr =
35     Forid of string
36
37  type stmt =
38     Block of stmt list
39     | Expr of expr
40     | Return of expr
41     | If of expr * stmt * stmt
42     | For of for_expr * for_expr * stmt
43     (* | For of expr * expr * stmt*)
44     | While of expr * stmt
45     | Print of expr
46     | Ifin of list_expr * list_expr * stmt * stmt
47
48  type var_decl = {
49     vtype : data_type;
50     vname : string;
51     vexpr : expr;
52  }
53
54  type func_decl = {
55     return : data_type;
56     fname : string;
57     formals : var_decl list;
58     fnlocals : var_decl list;
59     body : stmt list;
60  }
61
62
63  type program = var_decl list * func_decl list
```

1.4 SAST

```
1  type op_t = Add | Sub | Mult | Div | Equal | Neq | Less | Leq
2          | Greater | Geq | And | Or | StrEqual | StrNeq | StrAdd
3
4  type sep_t = Comma
5
6  type data_type_t = PathType | StrType | IntType | BoolType | VoidType | ListType
7
8  type pathattr_type_t = Pathname | Pathcreated | Pathkind | Pathext
9
10 type list_expr_t =
11     ListId of string * string
12   | ListItemInt of int
13   | ListItemStr of string
14
15 type items_t =
16     Item of expr_t
17   | Seq of expr_t * sep_t * items_t
18   | Noitem
19 and expr_t =
20     LitInt of int
21   | LitStr of string
22   | Id of string
23   | Binop of expr_t * op_t * expr_t
24   | Assign of string * expr_t
25   | Call of string * expr_t list
26   | Copy of expr_t * expr_t
27   | Move of expr_t * expr_t
28   | List of items_t
29   | ListAppend of string * list_expr_t
30   | ListRemove of string * list_expr_t
31   | Pathattr of string * pathattr_type_t
32   | Noexpr
33
34 type for_expr_t =
35     Forid of string
36
37 type stmt_t =
38     Block of stmt_t list
39   | Expr of expr_t
40   | Return of expr_t
41   | If of expr_t * stmt_t * stmt_t
42   (*| For of expr_t * expr_t * stmt_t *)
43   | For of for_expr_t * for_expr_t * stmt_t
44   | While of expr_t * stmt_t
45   | Print of expr_t * string
46   | Ifin of list_expr_t * list_expr_t * stmt_t * stmt_t
47
48 type var_decl_t = {
49     vtype : data_type_t;
50     vname : string;
51     vexpr : expr_t;
52 }
53
54 type func_decl_t = {
55     return : data_type_t;
56     fname : string;
57     formals : var_decl_t list;
58     fnlocals : var_decl_t list;
59     body : stmt_t list;
60 }
61
62 type program_t = var_decl_t list * func_decl_t list
```

1.5 Type Check

```
1 open Ast
2 open Symboltable
3
4 module StringMap = Map.Make(String)
5
6 let string_of_vtype = function
7   VoidType -> "void"
8   | IntType -> "int"
9   | StrType -> "string"
10  | BoolType -> "bool"
11  | PathType -> "path"
12  | ListType -> "list"
13
14 let get_sast_type = function
15   Ast.PathType -> Sast.PathType
16   | Ast.StrType -> Sast.StrType
17   | Ast.IntType -> Sast.IntType
18   | Ast.BoolType -> Sast.BoolType
19   | Ast.VoidType -> Sast.VoidType
20   | Ast.ListType -> Sast.ListType
21
22 let get_sast_pathattrtype = function
23   Ast.Pathname -> Sast.Pathname, "string"
24   | Ast.Pathcreated -> Sast.Pathcreated, "int"
25   | Ast.Pathkind -> Sast.Pathkind, "int"
26   | Ast.Pathext -> Sast.Pathext, "string"
27
28 let get_vtype env id =
29   (* find_variable method is from the symbol table *)
30   let t = find_variable id env in
31   if t = "" then raise (Failure ("undefined variable: " ^ id)) else t
32
33 let get_expr_type t1 t2 =
34   if t1 = "void" || t2 = "void" then raise (Failure ("cannot use void type inside expression")) else
35   if t1 = "string" || t2 = "string" then "string" else
36   if t1 = "int" && t2 = "int" then "int" else
37   if t1 = "bool" && t2 = "bool" then "bool" else
38   if t1 = "int" && t2 = "bool" then "int" else
39   if t1 = "bool" && t2 = "int" then "int" else
40   raise (Failure ("type error"))
```

```

41 let check_listexpr env = function
42 | Ast.ListId(id) ->
43   Sast.ListId(id, get_vtype env id), get_vtype env id
44 | Ast.ListItemInt(i) -> Sast.ListItemInt(i), "int"
45 | Ast.ListItemStr(s) -> Sast.ListItemStr(s), "string"
46
47
48 let match_oper e1 op e2 =
49   let expr_t = get_expr_type (snd e1) (snd e2) in
50   (match op with
51   | Ast.Add -> if expr_t = "int" then (Sast.Binop(fst e1, Sast.Add, fst e2), "int") else
52     if expr_t = "string" then (Sast.Binop(fst e1, Sast.StrAdd, fst e2), "string") else
53     raise (Failure ("type error"))
54 | Ast.Sub -> if expr_t = "int" then (Sast.Binop(fst e1, Sast.Sub, fst e2), "int") else
55   raise (Failure ("type error"))
56 | Ast.Mult -> if expr_t = "int" then (Sast.Binop(fst e1, Sast.Mult, fst e2), "int") else
57   raise (Failure ("type error"))
58 | Ast.Div -> if expr_t = "int" then (Sast.Binop(fst e1, Sast.Div, fst e2), "int") else
59   raise (Failure ("type error"))
60   (* equal and not equal have special case for string comparison
61   we may need to add SAST and Eqs and Neqs *)
62 | Ast.Equal -> if expr_t = "int" then (Sast.Binop(fst e1, Sast.Equal, fst e2), "bool") else
63   if expr_t = "string" then (Sast.Binop(fst e1, Sast.StrEqual, fst e2), "bool") else
64   raise (Failure ("type error in == "))
65 | Ast.Neq -> if expr_t = "int" then (Sast.Binop(fst e1, Sast.Neq, fst e2), "bool") else
66   if expr_t = "string" then (Sast.Binop(fst e1, Sast.StrNeq, fst e2), "bool") else
67   raise (Failure ("type error"))
68 | Ast.Less -> if expr_t = "int" then (Sast.Binop(fst e1, Sast.Less, fst e2), "bool") else
69   raise (Failure ("type error"))
70 | Ast.Leq -> if expr_t = "int" then (Sast.Binop(fst e1, Sast.Leq, fst e2), "bool") else
71   raise (Failure ("type error"))
72 | Ast.Greater -> if expr_t = "int" then (Sast.Binop(fst e1, Sast.Greater, fst e2), "bool") else
73   raise (Failure ("type error"))
74 | Ast.Geq -> if expr_t = "int" then (Sast.Binop(fst e1, Sast.Geq, fst e2), "bool") else
75   raise (Failure ("type error"))
76 | Ast.And -> if expr_t = "bool" then (Sast.Binop(fst e1, Sast.And, fst e2), "bool") else
77   raise (Failure ("type error in and"))
78 | Ast.Or -> if expr_t = "bool" then (Sast.Binop(fst e1, Sast.Or, fst e2), "bool") else
79   raise (Failure ("type error in or"))
80 )

```



```

81 let rec check_expr env = function
82   Ast.LitInt(i) -> Sast.LitInt(i), "int"
83   | Ast.LitStr(s) -> Sast.LitStr(s), "string"
84
85   | Ast.Id(id) ->
86     Sast.Id(id), (get_vtype env id)
87
88   | Ast.Binop(e1, op, e2) ->
89     match_oper (check_expr env e1) op (check_expr env e2)
90
91   | Ast.Assign(id, e) ->
92     let t = get_vtype env id in
93     Sast.Assign(id, (get_expr_with_type env e t)), "void"
94
95   | Ast.Call(func, el) ->
96     let args = find_function func env in (* return & arguments type list from definition *)
97     ( match args with
98       [] -> raise (Failure ("undefined function " ^ func))
99       | hd::tl -> let new_list = try List.fold_left2 check_func_arg [] (List.map (check_expr env) el) tl
100                    with Invalid_argument "arg" -> raise(Failure("unmatched argument list"))
101                    in Sast.Call(func, List.rev new_list ), hd )
102
103   | Ast.Move(e1, e2) ->
104     let e_t1 = check_expr env e1 in
105     let e_t2 = check_expr env e2 in
106     if snd e_t1 = "path" && snd e_t2 = "path"
107     then Sast.Move(fst e_t1, fst e_t2), "void"
108     else
109       raise(Failure("cannot use path function on non-path variables"))
110   | Ast.Copy(e1, e2) ->
111     let e_t1 = check_expr env e1 in
112     let e_t2 = check_expr env e2 in
113     if snd e_t1 = "path" && snd e_t2 = "path"
114     then Sast.Copy(fst e_t1, fst e_t2), "void"
115     else
116       raise(Failure("cannot use path function on non-path variables"))
117   | Ast.List(items) -> Sast.List(check_list_items env items), "list"
118   | Ast.ListAppend(id, item) -> let t1 = get_vtype env id in
119     let t2 = check_listexpr env item in
120     if not (t1 = "list")
121     then raise(Failure("Can append only to id of type list.))
122     else if ((snd t2) = "list")
123     then raise(Failure("Cannot append list to list.))
124     else
125       Sast.ListAppend( id, (fst t2)), "void"
126   | Ast.ListRemove(id, item) -> let t1 = get_vtype env id in
127     let t2 = check_listexpr env item in
128     if not (t1 = "list")
129     then raise(Failure("Can call remove only on type list.))
130     else if ((snd t2) = "list")
131     then raise(Failure("Cannot remove a list from list.))
132     else
133       Sast.ListRemove(id, (fst t2)), "void"
134   | Ast.Pathattr(id, e) ->
135     if not ((get_vtype env id) = "path")
136     then raise(Failure("cannot use path attributes on non-path variable " ^ id))
137     else
138       Sast.Pathattr(id, fst (get_sast_pathattrtype e)), snd (get_sast_pathattrtype e)
139   | Ast.Noexpr -> Sast.Noexpr, "void"
140
141 and check_list_items env = function
142   Ast.Item(e) ->let i,t = check_expr env e in
143     Sast.Item(i)
144   | Ast.Seq(e1, sep, e2) -> Sast.Seq(fst (check_expr env e1), Sast.Comma, (check_list_items env e2))
145   | Ast.Noitem -> Sast.Noitem
146
147 and get_expr_with_type env expr t =
148   let e = check_expr env expr in
149   if ((snd e) = "string" && t = "path") then (fst e)
150   else if ((snd e) = "int" && t = "bool") then (fst e)
151   else if not((snd e) = t) then raise (Failure ("type error")) else (fst e)
152
153 let check_forexpr env = function
154   Ast.Forid(id) -> Sast.Forid(id), get_vtype env id

```

```

155 let rec check_stmt env func = function
156   Ast.Block(stmt_list) -> (Sast.Block(check_stmt_list env func stmt_list)), env
157 | Ast.Expr(expr) -> (Sast.Expr(fst (check_expr env expr))), env
158 | Ast.Return(expr) -> let e = check_expr env expr in
159   if not(snd e = string_of_vtype func.return) then raise (Failure ("The return type doesn't match!"))
160   else (Sast.Return(fst e)), env
161 | Ast.If(expr, stmt1, stmt2) -> let e = check_expr env expr in
162   if not(snd e = "bool") then raise (Failure ("The type of the condition in If statement must be boolean!"))
163   else (Sast.If(fst e, fst (check_stmt env func stmt1), fst (check_stmt env func stmt2))), env
164 | Ast.IfIn(lexpr1, lexpr2, stmt1, stmt2) -> let e1 = check_listexpr env lexpr1 in
165   if (snd e1 = "list") then raise (Failure ("Cannot have list in list!"))
166   else let e2 = check_listexpr env lexpr2 in
167     if not(snd e2 = "list") then raise (Failure ("'\in' operator works with list type expression only!"))
168     else (Sast.IfIn(fst e1, fst e2, fst (check_stmt env func stmt1), fst (check_stmt env func stmt2))), env
169 | Ast.While(expr, stmt) -> let e = check_expr env expr in
170   if not (snd e = "bool") then raise (Failure ("The type of the condition in While statement must be boolean!"))
171   else (Sast.While(fst e, fst (check_stmt env func stmt))), env (* while() {} *)
172 | Ast.For(expr1, expr2, stmt) -> let e1 = check_forexpr env expr1 in let e2 = check_forexpr env expr2 in
173   if not (snd e1 = "path" && snd e2 = "path" ) then raise
174     (Failure("The type of the expression in a For statement must be path"))
175   else (Sast.For(fst e1, fst e2, fst (check_stmt env func stmt))), env
176 | Ast.Print(expr) -> let (expr, expr_type) = check_expr env expr in
177   (Sast.Print(expr , expr_type)), env
178
179 let rec check_expr env = function
180   Ast.LitInt(i) -> Sast.LitInt(i), "int"
181 | Ast.LitStr(s) -> Sast.LitStr(s), "string"
182
183 | Ast.Id(id) ->
184   Sast.Id(id), (get_vtype env id)
185
186 | Ast.Binop(e1, op, e2) ->
187   match_oper (check_expr env e1) op (check_expr env e2)
188
189 | Ast.Assign(id, e) ->
190   let t = get_vtype env id in
191   Sast.Assign(id, (get_expr_with_type env e t)), "void"
192
193 | Ast.Call(func, el) ->
194   let args = find_function func env in (* return & arguments type list from definition *)
195   ( match args with
196     [] -> raise (Failure ("undefined function " ^ func))
197   | hd::tl -> let new_list = try List.fold_left2 check_func_arg [] (List.map (check_expr env) el) tl
198     with Invalid_argument "arg" -> raise(Failure("unmatched argument list"))
199     in Sast.Call(func, List.rev new_list ), hd )
200   (* Need to add type checking for Move and Copy *)
201 | Ast.Move(e1, e2) ->
202   let e_t1 = check_expr env e1 in
203   let e_t2 = check_expr env e2 in
204   if snd e_t1 = "path" && snd e_t2 = "path"
205   then Sast.Move(fst e_t1, fst e_t2), "void"
206   else
207     raise(Failure("cannot use path function on non-path variables"))
208 | Ast.Copy(e1, e2) ->
209   let e_t1 = check_expr env e1 in
210   let e_t2 = check_expr env e2 in
211   if snd e_t1 = "path" && snd e_t2 = "path"
212   then Sast.Copy(fst e_t1, fst e_t2), "void"
213   else
214     raise(Failure("cannot use path function on non-path variables"))

```

```

215 | Ast.List(items) -> Sast.List(check_list_items env items), "list"
216 | Ast.ListAppend(id, item) -> let t1 = get_vtype env id in
217   let t2 = check_listexpr env item in
218   if not (t1 = "list")
219   then raise(Failure("Can append only to id of type list."))
220   else if ((snd t2) = "list")
221   then raise(Failure("Cannot append list to list."))
222   else
223     Sast.ListAppend( id, (fst t2)), "void"
224 | Ast.ListRemove(id, item) -> let t1 = get_vtype env id in
225   let t2 = check_listexpr env item in
226   if not (t1 = "list")
227   then raise(Failure("Can call remove only on type list."))
228   else if ((snd t2) = "list")
229   then raise(Failure("Cannot remove a list from list."))
230   else
231     Sast.ListRemove(id, (fst t2)), "void"
232 | Ast.Pathattr(id, e) ->
233   if not ((get_vtype env id) = "path")
234   then raise(Failure("cannot use path attributes on non-path variable " ^ id))
235   else
236     (* return type is string assuming path attributes will be treated that way *)
237     Sast.Pathattr(id, fst (get_sast_pathattrtype e)), snd (get_sast_pathattrtype e)
238 | Ast.Noexpr -> Sast.Noexpr, "void"
239
240 and check_list_items env = function
241   Ast.Item(e) -> let i, t = check_expr env e in
242     Sast.Item(i)
243 | Ast.Seq(e1, sep, e2) -> Sast.Seq(fst (check_expr env e1), Sast.Comma, (check_list_items env e2))
244 | Ast.Noitem -> Sast.Noitem
245
246 and get_expr_with_type env expr t =
247   let e = check_expr env expr in
248   (* added special case for the path variable *)
249   if ((snd e) = "string" && t = "path") then (fst e)
250   else if ((snd e) = "int" && t = "bool") then (fst e)
251   else if not((snd e) = t) then raise (Failure ("type error")) else (fst e)
252
253
254 let check_forexpr env = function
255   Ast.Forid(id) -> Sast.Forid(id), get_vtype env id

```

```

256 let rec check_stmt env func = function
257   Ast.Block(stmt_list) -> (Sast.Block(check_stmt_list env func stmt_list)), env
258 | Ast.Expr(expr) -> (Sast.Expr(fst (check_expr env expr))), env
259 | Ast.Return(expr) -> let e = check_expr env expr in
260   if not(snd e = string_of_vtype func.return) then raise (Failure ("The return type doesn't match!"))
261   else (Sast.Return(fst e)), env
262 | Ast.If(expr, stmt1, stmt2) -> let e = check_expr env expr in
263   if not(snd e = "bool") then raise (Failure ("The type of the condition in If statement must be boolean!"))
264   else (Sast.If(fst e, fst (check_stmt env func stmt1), fst (check_stmt env func stmt2))), env (* if() {} else{} *)
265 | Ast.IfIn(lexpr1, lexpr2, stmt1, stmt2) -> let e1 = check_listexpr env lexpr1 in
266   if (snd e1 = "list") then raise (Failure ("Cannot have list in list!"))
267   else let e2 = check_listexpr env lexpr2 in
268     if not(snd e2 = "list") then raise (Failure ("'\in' operator works with list type expression only!"))
269     else (Sast.IfIn(fst e1, fst e2, fst (check_stmt env func stmt1), fst (check_stmt env func stmt2))), env
270 | Ast.While(expr, stmt) -> let e = check_expr env expr in
271   if not (snd e = "bool") then raise (Failure ("The type of the condition in While statement must be boolean!"))
272   else (Sast.While(fst e, fst (check_stmt env func stmt))), env (* while() {} *)
273 | Ast.For(expr1, expr2, stmt) -> let e1 = check_forexpr env expr1 in let e2 = check_forexpr env expr2 in
274   if not (snd e1 = "path" && snd e2 = "path" ) then raise (Failure("The type of the expression in a For statement must be path"))
275   else (Sast.For(fst e1, fst e2, fst (check_stmt env func stmt))), env
276 | Ast.Print(expr) -> let (expr, expr_type) = check_expr env expr in
277   (Sast.Print(expr, expr_type)), env
278
279 and check_stmt_list env func = function
280   [] -> []
281 | hd::tl -> let s,e = (check_stmt env func hd) in s::(check_stmt_list e func tl)
282
283 let convert_to_sast_type x env =
284   let t = get_vtype env x.vname in
285   let s_expr =
286     if not (x.vexpr = Ast.Noexpr) then
287       get_expr_with_type env x.vexpr t
288     else Sast.Noexpr
289   in
290   {
291     Sast.vtype = get_sast_type x.vtype;
292     Sast.vname = x.vname;
293     Sast.vexpr = s_expr;
294   }
295
296 let check_formal env formal =
297   let ret = add_local formal.vname formal.vtype env in
298   if (string_of_vtype formal.vtype) = "void" then raise (Failure("cannot use void as variable type")) else
299   if StringMap.is_empty ret then raise (Failure ("local variable " ^ formal.vname ^ " is already defined"))
300   else let env = {locals = ret; globals = env.globals; functions = env.functions } in
301   convert_to_sast_type formal env, env
302
303 let rec check_formals env formals =
304   match formals with
305   [] -> []
306 | hd::tl -> let f, e = (check_formal env hd) in (f, e)::(check_formals e tl)
307
308 let check_local env local =
309   let ret = add_local local.vname local.vtype env in
310   if (string_of_vtype local.vtype) = "void" then raise (Failure("cannot use void as variable type")) else
311   if StringMap.is_empty ret then raise (Failure ("local variable " ^ local.vname ^ " is already defined"))
312   else let env = {locals = ret; globals = env.globals; functions = env.functions } in
313   convert_to_sast_type local env, env

```

```

314 let rec check_locals env locals =
315   match locals with
316   [] -> []
317   | hd::tl -> let l, e = (check_local env hd) in (l, e)::(check_locals e tl)
318
319 let check_function env func =
320   match List.hd (List.rev func.body) with
321   Return(_) ->
322     let env = {locals = StringMap.empty; globals = env.globals; functions = env.functions } in
323     (* ret is new env *)
324     let ret = add_function func.fname func.return func.formals env in
325     if StringMap.is_empty ret then raise (Failure ("function " ^ func.fname ^ " is already defined"))
326     else let env = {locals = env.locals; globals = env.globals; functions = ret } in
327     let f = check_formals env func.formals in
328     let formals = List.map (fun formal -> fst formal) f in
329
330     (* get the final env from the last formal *)
331     let l, env =
332       (match f with
333        [] -> let l = check_locals env func.fnlocals in
334              l, env
335        | _ -> let env = snd (List.hd (List.rev f)) in
336              let l = check_locals env func.fnlocals in
337              l, env
338       ) in
339     let fnlocals = List.map (fun fnlocal -> fst fnlocal) l in
340     (match l with
341      [] -> let body = check_stmt_list env func func.body in
342            { Sast.return = get_sast_type func.return;
343              Sast.fname = func.fname;
344              Sast.formals = formals;
345              Sast.fnlocals = fnlocals;
346              Sast.body = body
347            }, env
348      | _ -> let e = snd (List.hd (List.rev l)) in
349            let body = check_stmt_list e func func.body in
350            { Sast.return = get_sast_type func.return;
351              Sast.fname = func.fname;
352              Sast.formals = formals;
353              Sast.fnlocals = fnlocals;
354              Sast.body = body
355            }, e
356      )
357     | _ -> raise (Failure ("The last statement must be return statement"))
358
359 let rec check_functions env funcs =
360   match funcs with
361   [] -> []
362   | hd::tl -> let f, e = (check_function env hd) in f::(check_functions e tl)
363
364 let check_global env global =
365   if (string_of_vtype global.vtype) = "void" then raise (Failure("cannot use void as variable type"))
366   else let ret = add_global global.vname global.vtype env in
367   if StringMap.is_empty ret then raise (Failure ("global variable " ^ global.vname ^ " is already defined"))
368   (* update the env with globals from ret *)
369   else let env = {locals = env.locals; globals = ret; functions = env.functions } in
370   convert_to_sast_type global env, env
371
372 let rec check_globals env globals =
373   match globals with
374   [] -> []
375   | hd::tl -> let g, e = (check_global env hd) in (g, e)::(check_globals e tl)
376
377 let check_program (globals, funcs) =
378   let env = { locals = StringMap.empty;
379              globals = StringMap.empty;
380              functions = StringMap.empty }
381   in
382   let g = check_globals env globals in
383   let globals = List.map (fun global -> fst global) g in
384   match g with
385   [] -> (globals, (check_functions env (List.rev funcs)))
386   | _ -> 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

```
1  #!/bin/sh
2
3  if [ ! -f "c/libraries/liblist.a" ] || [ ! -f "c/libraries/libpath.a" ] ; then
4      cd c/libraries
5      make >> lib_msgs.txt
6      cd ../../
7  fi
8
9  if [ ! -f "preprocessor/./preprocessor" ]; then
10     cd preprocessor
11     make >> preproc_msgs.txt
12     cd ..
13 fi
14
15 if [ ! -f "./fdl" ]; then
16     make >> compiler_msgs.txt
17 fi
18
19 FDL="./fdl"
20 PRE="preprocessor/./preprocessor"
21
22 Compare() {
23     difference=$(diff -b $1 $2)
24     echo $difference
25     if [ "$difference" != "" ]; then
26         echo $difference > $3
27     fi
28 }
29
30 function compile() {
31     basename='echo $1 | sed 's/.*\\///
32             s/.fdl//''
33     reffile='echo $1 | sed 's/.fdl$//'
34     prepfile=${reffile}'.fdlp'
35     basedir="'echo $1 | sed 's/\\/[^\\]*$//'/'
36
37     testoutput='echo ${basedir}test_outputs/$basename.c.out'
38
39     echo "Preprocessing '$1'..."
40     $PRE $1 $prepfile && echo "Preprocessor for $1 succeeded"
41
42     echo "Compiling $prepfile ..."
43     $FDL $prepfile > "${reffile}.c" && echo "Ocaml to C of $1 succeeded"
44
45     if [ -f "${reffile}.c" ]; then
46         gcc -Ic/libraries -Lc/libraries -llist -lpath "${reffile}.c" -o "${reffile}" && echo "COMPILE of ${reffile}.c succeeded"
47     else
48         echo "Ocaml to C of $1 failed"
49         return
50     fi
51
52     rm -rf $prepfile
53
54     if [ -f "${reffile}" ]; then
55         eval ${reffile} > ${reffile}.generated.out
56         Compare ${testoutput} ${reffile}.generated.out ${reffile}.c.diff
57         rm -rf ${reffile}.generated.out
58         rm -rf ${reffile}.c
59         rm -rf ${reffile}
60     else
61         echo "C to binary of ${reffile}.c failed"
62     fi
63 }
64
65 files=sample_program/*.fdl
66
67 for file in $files
68 do
69     compile $file
70 done
```

2.2 Run FDL Script

```
1  #!/bin/sh
2
3  if [ ! -f "c/libraries/liblist.a" ] || [ ! -f "c/libraries/libpath.a" ] ; then
4      cd c/libraries
5      make >> lib_msgs.txt
6      cd ../../
7  fi
8
9  if [ ! -f "preprocessor/./preprocessor" ]; then
10     cd preprocessor
11     make >> preproc_msgs.txt
12     cd ..
13 fi
14
15 if [ ! -f "./fdl" ]; then
16     make >> compiler_msgs.txt
17 fi
18
19 # fdl executable
20 FDL="./fdl"
21 # preprocessor executable
22 PRE="./preprocessor/preprocessor"
23
24 function compileAndRun() {
25     basename='echo $1 | sed 's/.*\\///
26                               s/./fdl//''
27     reffile='echo $1 | sed 's/./fdl$//''
28     prepfile=$reffile'.fdlp'
29     basedir='echo $1 | sed 's/\\/[^\[]*$///''/'
30
31
32     $PRE $1 $prepfile
33
34     if [ ! -f $prepfile ]; then
35         echo "$prepfile does not exist"
36         return
37     fi
38
39     $FDL $prepfile > "${reffile}.c"
40
41     if [ -f "${reffile}.c" ]; then
42         gcc -Ic/libraries -Lc/libraries -llist -lpath -w -o "${reffile}" "${reffile}.c"
43     else
44         echo "Ocaml to C of $1 failed"
45         return
46     fi
47
48     if [ -f "${reffile}" ]; then
49         eval ${reffile}
50         rm -rf ${reffile}.fdlp
51         rm -rf ${reffile}.c
52         rm -rf ${reffile}
53     else
54         echo "C to binary of ${reffile}.c failed"
55     fi
56 }
57
58 if [ -f $1 ]; then
59     compileAndRun $1
60 else
61     echo "$1 doesnt exist"
62 fi
```


2.3 Clean All Script

```
1  #!/bin/sh
2
3  if [ -f "c/libraries/liblist.a" ] || [ -f "c/libraries/libpath.a" ]; then
4      cd c/libraries
5      make clean
6      cd ../..
7  fi
8
9  if [ -f "preprocessor/./preprocessor" ]; then
10     cd preprocessor
11     make clean
12     cd ..
13 fi
14
15 if [ -f "./fdl" ]; then
16     make clean
17 fi
```

3 Preprocessor

3.1 Makefile

```
1  CC = gcc
2  CXX = g++
3
4  INCLUDES =
5  CFLAGS = -g -Wall $(INCLUDES)
6  CXXFLAGS = -g -Wall $(INCLUDES)
7
8  LDFLAGS =
9  LDLIBS =
10
11 .PHONY: default
12 default: preprocessor
13
14 # header dependency
15 preprocessor: preprocessor.o
16
17 .PHONY: clean
18 clean:
19     rm -f *.o *.txt *~ a.out core preprocessor
20
21 .PHONY: all
22 all: clean default
```

3.2 Preprocessor

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <unistd.h>
5  #include <assert.h>
6  #include <ctype.h> /* For isspace(). */
7  #include <stddef.h> /* For size_t. */
8
9  #define MAX_BUFFER 4096
10
11 static void die(const char *message)
12 {
13     perror(message);
14     exit(1);
15 }
16
17 const char *getFileExtension(const char *fileName) {
18     const char *dot = strrchr(fileName, '.');
19     if(!dot || dot == fileName) return "";
20     return dot + 1;
21 }
22
23 void remove_whitespace(char *str) {
24     char *p;
25     size_t len = strlen(str);
26
27     for(p = str; *p; p ++, len --) {
28         while(isspace(*p)) memmove(p, p+1, len--);
29     }
30 }
31
32 int is_empty(const char *s) {
33     while (*s != '\0') {
34         if (!isspace(*s))
35             return 0;
36         s++;
37     }
38     return 1;
39 }
40
41
42 int main(int argc, char const *argv[])
43 {
44     if (argc != 3) {
45         fprintf(stderr, "%s\n", "usage: ./preprocessor <fdl file> <fdlp file>");
46         exit(1);
47     }
48     char *fileName = (char *) argv[1];
49     char *outputFileName = (char *) argv[2];
50
51     if (strcmp("fdl", getFileExtension(fileName)) != 0)
52     {
53         die("file extension must be fdl");
54     }
55     if (strcmp("fdlp", getFileExtension(outputFileName)) != 0)
56     {
57         die("output file extension must be fdlp");
58     }
59     FILE *input;
60     if ((input = fopen(fileName, "r")) == NULL) {
61         die("fopen() failed");
62     }
63     FILE *output;
64     if ((output = fopen(outputFileName, "w")) == NULL) {
65         die("fopen() failed");
66     }
67
68     char buffer[MAX_BUFFER];
```

```

69 while (fgets(buffer, sizeof(buffer), input) != NULL) {
70     size_t len = strlen(buffer) - 1;
71     if (buffer[len] == '\n') {
72         buffer[len] = '\0';
73     }
74     if (strstr(buffer, "*/") != NULL) {
75         fprintf(output, "%s\n", buffer);
76     }
77     else if (strstr(buffer, "/*") != NULL) {
78         fprintf(output, "%s\n", buffer);
79     }
80     else if (strstr(buffer, "def ") != NULL) {
81         fprintf(output, "%s {\n", buffer);
82     }
83     else if (strstr(buffer, "int ") != NULL) {
84         fprintf(output, "%s;\n", buffer);
85     }
86     else if (strstr(buffer, "path ") != NULL) {
87         fprintf(output, "%s;\n", buffer);
88     }
89     else if (strstr(buffer, "dict ") != NULL) {
90         fprintf(output, "%s;\n", buffer);
91     }
92     else if (strstr(buffer, "list ") != NULL) {
93         fprintf(output, "%s;\n", buffer);
94     }
95     else if (strstr(buffer, "string ") != NULL) {
96         fprintf(output, "%s;\n", buffer);
97     }
98     else if (strstr(buffer, "bool ") != NULL) {
99         fprintf(output, "%s;\n", buffer);
100    }
101    else if (strstr(buffer, "for ") != NULL) {
102        fprintf(output, "%s {\n", buffer);
103    }
104    else if ((strstr(buffer, "if (") != NULL || strstr(buffer, "if(") != NULL)
105              && (strstr(buffer, "then") != NULL)) {
106        fprintf(output, "%s {\n", buffer);
107    }
108    else if ((strstr(buffer, "if (") != NULL || strstr(buffer, "if(") != NULL)
109              && (strstr(buffer, "then") == NULL)) {
110        fprintf(output, "%s\n", buffer);
111    }
112    else if (strstr(buffer, "then") != NULL) {
113        fprintf(output, "%s {\n", buffer);
114    }

```

```

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

```
1  CC = gcc
2  CXX = g++
3
4  INCLUDES = -I libraries/
5
6  CFLAGS = -g -Wall $(INCLUDES)
7  CXXFLAGS = -g -Wall $(INCLUDES)
8
9  LDFLAGS = -g -L libraries/
10
11 LDLIBS = -lstdc++ -lpath
12
13 stat_calls: stat_calls.o
14
15 stat_calls.o: stat_calls.c
16
17 .PHONY: clean
18 clean:
19     rm -f *.o *.txt a.out core stat_calls
20
21 .PHONY: all
22 all: clean stat_calls
```

4.2 Lists

4.2.1 List Header

```
1  #ifndef _LIST_H_
2  #define _LIST_H_
3
4  enum fdl_type { fdl_str, fdl_path, fdl_int, fdl_bool };
5
6  struct Node {
7      enum fdl_type type;
8      union {
9          int int_item;
10         int bool_item;
11         char *string_item;
12         char *path_item;
13     };
14     struct Node *next;
15 };
16
17 struct List {
18     struct Node *head;
19 };
20
21 struct Node *createIntNode(int data, enum fdl_type type);
22 struct Node *createStrNode(char *data, enum fdl_type type);
23
24 static inline void initList(struct List *list)
25 {
26     list->head = 0;
27 }
28
29 static inline int isEmptyList(struct List *list)
30 {
31     return (list->head == 0);
32 }
33
34 void addFront(struct List *list, struct Node *node);
35 void traverseList(struct List *list, void (*f)(struct Node *));
36 void printNode(struct Node *node);
37 int findNode(struct List *list, struct Node *node1);
38 void removeNode(struct List *list, struct Node *node1);
39 struct Node popFront(struct List *list);
40 void removeAllNodes(struct List *list);
41 void addAfter(struct List *list, struct Node *prevNode, struct Node *newNode);
42 void reverseList(struct List *list);
43 void addBack(struct List *list, struct Node *newNode);
44 void loadDirectoryToList(char *path, struct List *subPath);
45
46 #endif
```

4.2.2 List Implementation

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "list.h"
5  #include "dirent.h"
6
7  void loadDirectoryToList(char *path, struct List *subPath){
8      char *buffer;
9      DIR *dir;
10     struct dirent *ent;
11     int len;
12     if ((dir = opendir (path)) != NULL) {
13         /* print all the files and directories within directory */
14         while ((ent = readdir (dir)) != NULL) {
15             len = strlen(path) + strlen(ent->d_name) + 2;
16             buffer = (char *)malloc(sizeof(char)*len);
17             //printf("%s\n",ent->d_name);
18             strcpy(buffer, path);
19             strcat(buffer, "/");
20             strcat(buffer, ent->d_name);
21             struct Node * node = createStrNode(buffer, fdl_path);
22             addBack(subPath, node);
23             //buffer = "\0";
24         }
25         closedir (dir);
26     } else {
27         /* could not open directory */
28         perror ("");
29         exit(0);
30     }
31 }
32
33 struct Node *createIntNode(int data, enum fdl_type type) {
34     struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
35     if(newNode == NULL){
36         printf("Could not create new node!\n");
37         exit(1);
38     }
39
40     newNode->type = type;
41     newNode->next = NULL;
42     switch(newNode->type){
43         case fdl_int: newNode->int_item = data; break;
44         case fdl_bool: newNode->bool_item = data; break;
45         default: break;
46     }
47     return newNode;
48 }
49
50 struct Node *createStrNode(char *data, enum fdl_type type) {
51     struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
52     if(newNode == NULL){
53         printf("Could not create new node!\n");
54         exit(1);
55     }
56
57     newNode->type = type;
58     newNode->next = NULL;
59     switch(newNode->type){
60         case fdl_str: newNode->string_item = data; break;
61         case fdl_path: newNode->path_item = data; break;
62         default: break;
63     }
64     return newNode;
65 }
66
67 void addFront(struct List *list, struct Node *node)
68 {
69     node->next = list->head;
70     list->head = node;
71 }
```



```

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

```
1  #ifndef _PATH_H_
2  #define _PATH_H_
3
4  char* getName(char *path, char *output);
5  int checkValid(char *path);
6  int getCreatedAt(char *path);
7  int getPathType(char *path);
8  int isDir(char *path);
9  char* getPathName(char *path);
10 int copyFile(char* src, char *dest);
11 int moveFile(char* src, char *dest);
12 char* getExtension(char *path);
13 char* stringConcat(char *str1, char *str2);
14
15 #endif
```

4.3.2 Path Implementation

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "sys/stat.h"
5  #include "time.h"
6  #include<libgen.h>
7
8  // test function
9  char* getName(char *path, char *output){
10     char *dirc, *basec, *bname, *dname;
11
12     dirc = strdup(path);
13     basec = strdup(path);
14     dname = dirname(dirc);
15     bname = basename(basec);
16     //printf("dirname=%s, basename=%s\n", dname, bname);
17     strcpy(output, dname);
18     return output;
19 }
20
21 int checkValid(char *path){
22     /* testing the stat sys call for files and directories */
23     struct stat info;
24     if (stat(path, &info) != 0)
25         return 0;
26     else
27         // can be valid directory or file
28         return S_ISDIR(info.st_mode) ? 1 : S_ISREG(info.st_mode);
29 }
30
31 // returns -1 in case of invalid path
32 int getCreatedAt(char *path){
33     if(checkValid(path)){
34         struct stat info;
35         stat(path, &info);
36
37         return (int) info.st_birthtime;
38     }else
39         return -1;
40 }
41
42 // Directory 1, File 0, invalid path -1
43 int getPathType(char *path){
44     if(checkValid(path)){
45         struct stat info;
46         stat(path, &info);
47
48         return S_ISDIR(info.st_mode);
49     }else
50         return -1;
51 }
52
53 int isDir(char *path){
54     if(checkValid(path)){
55         struct stat info;
56         stat(path, &info);
57
58         return S_ISDIR(info.st_mode);
59     }else
60         return -1;
61 }
62
63 // get the last directory or filename
64 char* getPathName(char* path){
65     if(checkValid(path)){
66         char *basec = strdup(path);
67         char *bname = basename(basec);
68         return bname;
69     }else
70         return NULL;
71 }
72 }
```

```

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 }

```