ChemLAB

COMS W4115 - Programming Languages & Translators

Professor Stephen Edwards

Alice Chang (avc2120) Gabe Lu (ggl2110) Martin Ong (mo2454)

December 14, 2014

Contents

Introduction

ChemLab is a language that will allow users to conveniently manipulate chemical elements. It can be used to solve chemistry and organic chemistry problems including, but not limited to, stoichiometeric calculations, oxidation-reduction reactions, acid-base reactions, gas stoichiometry, chemical equilibrium, thermodynamics, stereochemistry, and electrochemistry. It may also be used for intensive study of a molecule's properties such as chirality or aromaticity. These questions are mostly procedural and there is a general approach to solving each specific type of problem. For example, to determine the molecular formula of a compound: 1) use the mass percents and molar mass to determine the mass of each element present in 1 mole of compound 2) determine the number of moles of each element present in 1 mole of compound. Albeit these problems can generally be distilled down to a series of plug-and-chug math calculations, these calculations can become extremely tedious to work out by hand as molecules and compounds become more complex (imagine having to balance a chemical equation with Botox: $C_{6760}H_{10447}N_{1743}O_{2010}S_{32}$). Our language can be used to easily create programs to solve such problems through the use of our specially designed data types and utilities.

Chapter 1

Language Tutorial

1.1 Program Execution

To compile a `.chem` program, simply use the compilation shell script with your `.chem` file as the only argument. `./compile.sh yourProgram.chem`

After compilation, if there are no errors, there will be a Java program that gets created. One can the compile the Java program into an executable using `javac`.

1.2 Variables

Variables in ChemLAB must be declared as a specific type. To use a variable, declare the type of the variable, and assign it to the value that you want like this:

```
int myNum = 5;
String hello = "World";
```

1.3 Control Flow

ChemLAB supports "if/else" statements:

```
if(10>6){
print("inside the if");
else{
print("inside the else");
```

```
}
```

ChemLAB supports "while loops":

```
while(i > 0){
print(i);
i = i-1;
}
```

1.4 Functions

Functions are the basis of ChemLAB. Functions can be passed any amount of parameters and are declared using the function keyword. The parameters within a function declaration must have type specifications.

This is a function that takes in two parameters:

```
function add (int a, int b){
return a+b;
}
```

This is a function that takes in no parameters:

```
function noParam(){
print("Hello World");
return 1;
}
```

1.5 Printing to stdout

To print to stdout, simply use the built-in function "print"

```
print(6);
print("Hello World");
```

Chapter 2

Language Reference Manual

Chapter 3

Project Plan

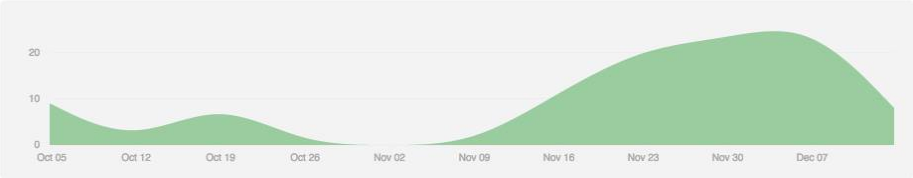
Like any project, careful planning and organization is paramount to the success of the project. More importantly however, is the methodical execution of the plan. Although we originally developed a roadmap for success as well as implemented a number of project management systems, we did not follow the plan as intended. This section outlines our proposed plans for making ChemLAB happen and the actual process that we went through.

3.1 Proposed Plan

We had originally planned to use the waterfall model in our software development process in which we would first develop a design for our language, followed by implementation, and finally testing. The idea was for all team members to dedicate complete focus to each stage in the project. Especially since we only had three members on our team, our roles were not as distinct and everyone had the chance to work, at least in some capacity, in all the roles. We intended to meet consistently each week on for at least two hours. During our meetings, each member was suppose to give an update about what he or she had been working on the past week as well as plans for the upcoming week and any challenges he or she faced that required the attention of the rest of the group. To help facilitate communication and the planning of meetings, we used Doodle to vote on what times were best for meetings. Also, in order to improve team dynamics, we planned to meet at least once every two weeks outside the context of school in order to hang out and have fun. Development would occur mostly on Mac OS and Windows 7, using the latest versions of OCaml, Ocamllex, and OCamlyacc for the compiler. We used Github for version control and makefiles to ease the work of compiling and testing code. The project timeline that we had laid out at the beginning was as follows:

- Sept 24th: Proposal Due Date
- Oct 2nd: ChemLAB syntax roughly decided upon
- Oct 23th: Scanner/Parser/AST unambiguous and working
- Oct 27th: LRM Due Date
- Nov 9th: Architectural design finalized
- Dec 5th: Compile works, all tests passed
- Dec 12th: Project report and slides completed
- Dec 17th: Final Project Due Date

3.2 What Actually Happened



This graph was pulled from Github reflecting the number of commits being made over the span of this semester. Due to schedule conflicts and a false sense of security, we did not start intensely working on the project until after Thanksgiving break. Since we did not coordinate the development of the Scanner, AST, and parser with the writing of the LRM, our language did not have as concrete a structure as we had hoped. Furthermore, we did not have enough time to implement some of the features in our language such as object-orientation or more built-in functions. As we were developing the software, we did make sure to allow testing at all steps in the design process. In the test script, we had identifiers for how far in the compilation process we wanted the program to run. Thus, we were able to maintain testing capabilities even before all of our code was ready. We discuss the testing procedure in more detail in a subsequent section.

3.3 Team Responsibilities

This subsection describes the contributions made by each team member:

- Project Proposal - Gabriel L/Alice C/Martin O

- Scanner - Gabriel L
- AST - Alice C/Gabriel L/Martin O
- Parser - Alice C/Martin O
- LRM - Gabriel L
- Code Generation - Alice C
- Semantic Analyzer -Gabriel L/Martin O
- Testing - Martin O
- Final Report - Gabriel L/Martin O

3.4 Project Log

We add later

Chapter 4

Architectural Design

Chapter 5

Test Plan

Chapter 6

Lessons Learned

Appendix A

Code Listing

../ast.ml

```
1  type operator = Add | Sub | Mul | Div | Equal | Neq | Lt | Leq | Gt | Geq
2  type re = And | Or
3  type bool = True | False
4  type data_type = IntType | BooleanType | StringType | DoubleType |
       ElementType | MoleculeType | EquationType
5
6  type variable =
7  Var of string
8
9  type expr =
10     Binop of expr * operator * expr
11   | Brela of expr * re * expr
12   | Int of int
13   | String of string
14   | Boolean of bool
15   | Double of float
16   | Asn of string * expr
17   | Equation of string * variable list * variable list
18   | Concat of expr * expr
19   | Seq of expr * expr
20   | Print of expr
21   | List of expr list
22   | Call of string * expr list
23   | Access of expr * string
24   | Draw of string * int * int * int * int * int * int * int * int
25   | Null
26   | Noexpr
27
28 type stmt =
29     Block of stmt list
30   | Expr of expr
```

```
31    | Return of expr
32    | If of expr * stmt * stmt
33    | For of expr * expr * expr * stmt
34    | While of expr * stmt
35    | Print of expr
36
37  type variable_decl = {
38    vname : string;
39    vtype : data_type;
40  }
41
42  type element_decl = {
43    name : string;
44    mass : int;
45    electrons : int;
46    charge : int;
47  }
48
49  type molecule_decl = {
50    mname : string;
51    elements: variable list;
52  }
53
54  type rule =
55      Balance of string
56      | Mass of string
57
58  type par_decl = {
59    paramname : string; (* Name of the variable *)
60    paramtype : data_type; (* Name of variable type *)
61  }
62
63  type func_decl = {
64    fname : string;
65    formals : par_decl list;
66    locals: variable_decl list;
67    elements : element_decl list;
68    molecules : molecule_decl list;
69    rules : rule list;
70    body : stmt list;
71  }
72
73  (* type program = {
74    gdecls : var_decl list;
75    fdecls : func_decl list
76  }
77   *)
78  type program = func_decl list
```

../scanner.mll

```
{ open Parser }

rule token = parse
    [' ' '\t' '\r' '\n']          { token lexbuf }
  | "/*"                        { comment lexbuf }
  | '('                           { LPAREN }
  | ')'                         { RPAREN }
  | '['                         { LBRACKET }
  | ']'                         { RBRACKET }
  | '{'                         { LCURLY }
  | '}'                         { RCURLY }
  | '"'                         { STRINGDECL }
  | ';'                                 { SEMI }
  | ':'                                 { COLON }
  | ','                                 { COMMA }
  | '.'                                 { ACCESS }
  | '+'                                 { PLUS }
  | '-'                                 { MINUS }
  | '*'                                 { TIMES }
  | '/'                                 { DIVIDE }
  | '%'                                 { MOD }
  | '='                                 { ASSIGN }
  | "->"                        { ARROW }
  | '^'                                 { CONCAT }
  | "=="                                { EQ }
  | "!="                                { NEQ }
  | '<'                                 { LT }
  | "<="                                { LEQ }
  | '>'                                 { GT }
  | ">="                                { GEQ }
  | "&&"                                { AND }
  | "||"                                { OR }
  | '!'                                 { NOT }
  | "if"                          { IF }
  | "else"                        { ELSE }
  | "while"                       { WHILE }
  | "for"                       { FOR }
  | "int"                         { INT }
  | "double"                        { DOUBLE }
  | "string"                      { STRING }
  | "boolean"                       { BOOLEAN }
  | "element"                       { ELEMENT }
  | "molecule"                      { MOLECULE}
  | "equation"                      { EQUATION }
  | "Balance"                     { BALANCE }
  | "mass"     as attr              { ATTRIBUTE(attr) }
  | "charge"     as attr               { ATTRIBUTE(attr) }
  | "electrons" as attr                { ATTRIBUTE(attr) }
  | "function"                      { FUNCTION }
```

```
50    | "object"                    { OBJECT }
51    | "return"                    { RETURN }
52    | "true"                    { BOOLEAN_LIT(true) }
53    | "false"                   { BOOLEAN_LIT(false) }
54    | "print"                   { PRINT }
55    | "Call"                  { CALL }
56    | "Draw"                  { DRAW }
57    | ['0'-'9']+ as lxm             { INT_LIT(int_of_string lxm) }
58    | ('0' | ['1'-'9']+['0'-'9']*)(['.']['0'-'9']+)? as lxm { DOUBLE_LIT(
        float_of_string lxm) }
59    | ['A'-'Z' 'a'-'z' '0'-'9']+ as lxm   { ID(lxm)}
60    | '"' [^'"']* '"'  as lxm         { STRING_LIT(lxm) }
61    | ['A'-'Z' ]['a'-'z']* as lxm     { ELEMENT_LIT(lxm)}
62    | (['A'-'Z']['a'-'z']* ['0'-'9']*)+ as lxm    { MOLECULE_LIT(lxm)}
63    | eof                             { EOF }
64    | _ as char                { raise (Failure("illegal character " ^
65                          Char.escaped char)) }
66
67  and comment = parse
68      "*/"          { token lexbuf }
69    | _           { comment lexbuf }
```

../parser.mly

```
1   %{ open Ast
2     let parse_error s = (* Called by parser on error *)
3       print_endline s;
4       flush stdout
5   %}
6
7   %token SEMI LPAREN RPAREN LBRACKET RBRACKET LCURLY RCURLY COMMA STRINGDECL
        COLON ACCESS CONCAT NOT OBJECT ARROW
8   %token PLUS MINUS TIMES DIVIDE MOD PRINT ASSIGN
9   %token EQ NEQ LT LEQ GT GEQ EQUAL
10
11  %token RETURN IF ELSE FOR WHILE INT DOUBLE STRING BOOLEAN ELEMENT MOLECULE
        EQUATION FUNCTION
12  %token INT DOUBLE STRING BOOLEAN ELEMENT MOLECULE EQUATION FUNCTION
13  %token CALL ACCESS DRAW
14  %token BALANCE MASS CHARGE ELECTRONS
15  %token AND OR
16  %token INT BOOLEAN STRING DOUBLE
17  %token <string> DATATYPE ATTRIBUTE
18  %token <bool> BOOLEAN_LIT
19  %token <string> ELEMENT_LIT
20  %token <string> MOLECULE_LIT
21  %token <string> STRING_LIT
22  %token <string> ID
23  %token <int> INT_LIT
24  %token <float> DOUBLE_LIT
```

```
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left ACCESS
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE

%start program
%type <Ast.program> program

%%
program:
  { [] }
  | program fdecl                                    { ($2 :: $1) }

id:
  ID                                               { $1 }
  | STRING_LIT                                        { $1 }
  | ELEMENT_LIT                                       { $1 }
  | MOLECULE_LIT                                      { $1 }

var:
  id {Var($1)}


vdecl:
  datatype ID SEMI
  { { vname = $2;
    vtype = $1;
  } }

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

stmt:
    expr SEMI                          { Expr($1) }
  | RETURN expr SEMI                       { Return($2)}
  | LCURLY stmt_list RCURLY                  { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE       { If($3, $5, Block([]) ) }
  | IF LPAREN expr RPAREN stmt ELSE stmt         { If($3, $5, $7) }
  | FOR LPAREN expr SEMI expr SEMI expr RPAREN stmt   { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt                 { While($3, $5) }
  | PRINT expr SEMI                      { Print($2)}
```

```
75
76  stmt_list:
77    /* nothing */ { [] }
78    | stmt_list stmt { ($2 :: $1) }
79
80  datatype:
81      INT      { IntType }
82    | BOOLEAN    { BooleanType }
83    | STRING   { StringType }
84    | DOUBLE   { DoubleType }
85
86  expr:
87    INT_LIT                                  { Int($1) }
88    | id                                    { String($1) }
89    | PRINT expr SEMI { Print($2)}
90    | EQUATION id LCURLY element_list ARROW element_list RCURLY   { Equation($2
        , $4, $6) }
91    | expr PLUS expr                              { Binop($1, Add, $3) }
92    | expr MINUS expr                               { Binop($1, Sub, $3) }
93    | expr TIMES expr                               { Binop($1, Mul, $3) }
94    | expr DIVIDE expr                              { Binop($1, Div, $3) }
95    | expr LT expr                                { Binop($1, Lt, $3) }
96    | expr GT expr                                { Binop($1, Gt, $3) }
97    | expr LEQ expr                               { Binop($1, Leq, $3) }
98    | expr AND expr                                      { Brela($1, And, $3) }
99    | expr OR expr                                       { Brela($1, Or, $3) }
100   | id ASSIGN expr                              { Asn($1, $3) }
101   | expr CONCAT expr                              { Concat($1, $3) }
102   | CALL id LPAREN actuals_opt RPAREN                    { Call($2, $4) }
103   | expr ACCESS ATTRIBUTE                          { Access($1, $3) }
104   | DRAW LPAREN STRING_LIT COMMA INT_LIT COMMA INT_LIT COMMA INT_LIT COMMA
        INT_LIT COMMA INT_LIT COMMA INT_LIT COMMA INT_LIT COMMA INT_LIT RPAREN
        { Draw($3, $5, $7, $9, $11, $13, $15, $17, $19) }
105
106
107
108 edecl:
109   ELEMENT id LPAREN INT_LIT COMMA INT_LIT COMMA INT_LIT RPAREN SEMI
110   {{
111     name = $2;
112     mass = $4;
113     electrons = $6;
114     charge = $8
115
116   }}
117
118 edecl_list:
119                          { [] }
120    | edecl_list edecl       { List.rev ($2 :: $1)}
121
```

```
122
123 mdecl:
124   MOLECULE id LCURLY element_list RCURLY SEMI
125   {{
126     mname = $2;
127     elements = $4;
128   }}
129
130 mdecl_list:
131     { [] }
132   | mdecl_list mdecl        { ($2 :: $1) }
133
134 element_list:
135   var               { [$1] }
136   | element_list COMMA var    { ($3 :: $1)}
137
138  rule:
139   BALANCE LPAREN id RPAREN SEMI {Balance($3)}
140
141
142  rule_list:
143   {[]}
144   | rule_list rule        { ($2 :: $1)}
145
146 formals_opt:
147   /* nothing */         { [] }
148   | formal_list         { List.rev $1 }
149
150 formal_list:
151   param_decl            { [$1] }
152   | formal_list COMMA param_decl  { $3 :: $1 }
153
154 actuals_opt:
155     /* nothing */         { [] }
156   | actuals_list          { List.rev $1 }
157
158 actuals_list:
159     expr                        { [$1] }
160   | actuals_list COMMA expr     { $3 :: $1 }
161
162 param_decl:
163   datatype id
164     { { paramname = $2;
165       paramtype = $1 } }
166
167 fdecl:
168   FUNCTION id LPAREN formals_opt RPAREN LCURLY vdecl_list edecl_list
        mdecl_list rule_list stmt_list RCURLY
169   { {
170     fname = $2;
```

```
    formals = $4;
    locals = List.rev $7;
    elements =  List.rev $8;
    molecules = List.rev $9;
    rules =  List.rev $10;
    body = List.rev $11
  } }
```

../semantic.ml

```
open Ast
open Str

type env = {
  mutable functions : func_decl list;
}

let function_equal_name name = function
  func-> func.fname = name

let function_fparam_name name = function
  par -> par.paramname = name

let function_var_name name = function
  variable -> variable.vname = name

(* Checks whether a function has been defined duplicately *)
let function_exist func env =
  let name = func.fname in
     try
       let _ = List.find (function_equal_name name) env.functions in
         let e = "Duplicate function: "^ name ^" has been defined more than
             once" in
           raise (Failure e)
    with Not_found -> false


(*Checks if function has been declared*)
let exist_function_name name env = List.exists (function_equal_name name) env
    .functions


let get_function_by_name name env =
  try
       let result = List.find (function_equal_name name) env.functions in
            result
  with Not_found -> raise(Failure("Function "^ name ^ " has not been declared
      !"))


```

```
38 let get_formal_by_name name func =
39   try
40     let result = List.find(function_fparam_name name) func.formals in
41       result
42   with Not_found -> raise(Failure("Formal Param" ^ name ^ " has not been
       declared!"))
43
44 let get_variable_by_name name func =
45   try
46     let result = List.find(function_var_name name) func.locals in
47     result
48   with Not_found -> raise(Failure("Local Variable " ^ name ^ "has not been
       declared!"))
49
50
51 let count_function_params func = function
52   a -> let f count b =
53   if b = a
54     then count+1
55     else count
56 in
57   let count = List.fold_left f 0 func.formals in
58     if count > 0
59       then raise (Failure("Duplicate parameter in function " ^ func.fname))
60       else count
61
62
63 let count_function_variables func = function
64   a -> let f count b =
65   if b = a
66     then count+1
67     else count
68 in
69   let count = List.fold_left f 0 func.locals in
70     if count > 0
71       then raise (Failure("Duplicate variable in function " ^ func.fname))
72       else count
73
74 (*Determines if a formal paramter with the given name  fpname  exits in
     the given function*)
75
76 let exists_formal_param func fpname =
77 try
78  List.exists (function_fparam_name fpname) func.formals
79 with Not_found -> raise (Failure ("Formal Parameter " ^ fpname ^ " should
     exist but was not found in function " ^ func.fname))
80
81
82 (*Determines if a variable declaration with the given name  vname  exists
      in the given functioin*)
```

```
83
84  let exists_variable_decl func vname =
85  try
86   List.exists (function_var_name vname) func.locals
87  with Not_found -> raise (Failure ("Variable " ^ vname ^ " should exist but
        was not found in function " ^ func.fname))
88
89
90
91
92  let dup_param_name func fpname =
93    let name = func.formals in
94      try
95        List.find (function name -> name.paramname = fpname.paramname ) name
96    with Not_found -> raise (Failure ("Duplicate param names"))
97
98
99
100 let get_fparam_type func fpname =
101   let name = func.formals in
102     try
103       let fparam = List.find(function_fparam_name fpname) name in
104         fparam.paramtype
105     with Not_found -> raise (Failure ("Formal param should exist but not
          found"))
106
107
108 (*given variable name, get type*)
109 let get_var_type func vname =
110   let name = func.locals in
111     try
112       let var = List.find(function_var_name vname) name in
113         var.vtype
114     with Not_found -> raise (Failure ("Variable should exist but not found"))
115
116 (*
117 let param_exist func =
118   let name = func.formals in
119   try
120     let _ = List.iter (fun f -> List.find (exists_formal_param func f) ) name
           in
121       let e = "Duplicate param: "^ name ^"has been defined more than once" in
122         raise (Failure e)
123   with Not_found -> false
124
125 let get_fparam_type func fpname =
126     try
127       let fparam =
128 *)
129
```

```
130 (*Determines if the given identifier exists*)
131 let exists_id name func = (exists_variable_decl func name) || (
        exists_formal_param func name)
132
133 (*see if there is a function with given name*)
134 let find_function func env =
135  try
136  let _ = List.find (function_equal_name func) env.functions in
137  true (*return true on success*)
138  with Not_found -> raise Not_found
139
140 let is_int s =
141  try ignore (int_of_string s); true
142  with _ -> false
143
144 let is_float s =
145  try ignore (float_of_string s); true
146  with _ -> false
147
148 let is_letter s = string_match (regexp "[A-Za-z]") s 0
149
150 let is_string s = string_match (regexp "\".*\"") s 0
151
152 let is_string_bool = function "true" -> true | "false" -> true | _ -> false
153
154 let rec is_num func = function
155     Int(_) -> true
156   | Double(_) -> true
157   | Binop(e1,_,e2) -> (is_num func e1) && (is_num func e2)
158   | _ -> false
159
160 let rec is_boolean func = function
161   Boolean(_) -> true
162   | _ -> false
163
164 (*check if variable declation is valid*)
165
166 (*
167
168 let valid_vdecl func =
169   let _ = List.map (function func.locals) ->
170   let e = "Invalid variable declaration for '" ^ nm ^ "' in compute function
        " ^ func.fname ^ "\n" in
171           let be = e ^ "The only allowed values for initializing boolean
                variables are 'true' and 'false.' \\n" in
172             match vtype with
173               "Int"  -> if is_string value then true else raise (Failure e)
174             | "Double"  -> if is_float value then true else raise (Failure e)
175             | "String"     -> if is_int value then true else raise (Failure e
                  )
```

```
176             | "Boolean" -> if is_string_bool value then true else raise (
                    Failure be)) func.locals
177             in
178               true
179
180 *)
181
182 let rec get_expr_type e func =
183   match e with
184     | String(s) -> StringType
185     | Int(s) -> IntType
186     | Double(f) -> DoubleType
187     | Boolean(b) -> BooleanType
188     | Binop(e1,op,e2) -> let t1 = get_expr_type e1 func and t2 =
          get_expr_type e2 func in
189       begin
190         match t1, t2 with
191           DoubleType, DoubleType -> DoubleType
192         | IntType, IntType -> IntType
193         | _,_ -> raise (Failure "Invalid types for binary expresion")
194       end
195     | Brela(e1, re, e2) -> let t1 = get_expr_type e1 func and t2 =
          get_expr_type e2 func in
196       begin
197         match t1, t2 with
198           BooleanType, BooleanType -> BooleanType
199         | _,_ -> raise (Failure "Invalid type for AND, OR expression")
200       end
201     | Asn(expr, expr2) -> get_expr_type expr2 func
202     | Equation (s, vlist, vlist2) -> EquationType
203     | Concat(s, s2) -> let s_type = get_expr_type s func in
204       let s2_type = get_expr_type s2 func in
205         begin
206           match s_type, s2_type with
207             StringType, StringType -> StringType
208           | _,_ -> raise (Failure "concatentation needs to be with two
                      strings")
209         end
210     | _ -> raise( Failure("!!! Need to implement in get_expr_type: Seq, List,
            Call, Null, Noexpr !!!") )
211
212 let rec valid_expr (func : Ast.func_decl) expr env =
213   match expr with
214     Int(_) -> true
215   | Double(_) ->  true
216   | Boolean(_) -> true
217   | String(_) -> true
218   | Binop(e1,_,e2) -> (is_num func e1) && (is_num func e2)
219   | Brela (e1,_,e2) -> (is_boolean func e1) && (is_boolean func e2)
220   | Asn(id, expr2) ->
```

```
221     begin
222       let t1 = get_var_type func id and t2 = get_expr_type expr2 func in
223         match t1,t2 with
224           StringType, StringType -> true
225         | IntType, IntType -> true
226         | DoubleType, DoubleType -> true
227         | ElementType, ElementType -> true (*allow int to double conversion*)
228         | MoleculeType, MoleculeType -> true
229         | EquationType, EquationType -> true
230         | _,_ -> raise(Failure ("DataTypes do not match up in an assignment
                expression to variable "))
231     end
232   | _ -> raise( Failure("!!! Need to implement in valid_expr: Equation,
        Concat, Seq, List, Call, Null, Noexpr !!!") )
233
234 (*Print(e1) ->
235     let t1 = get_expr_type expr func in
236       match t1 with
237         "String" -> true
238       |   "int" -> true
239       |   "double" -> true
240       |   "boolean" -> true
241       |   "element" -> true
242       |   "molecule" -> true
243       |   "equation" -> true
244       |   _ -> raise(Failure("Can't print type"))*)
245
246
247
248 let has_return_stmt list =
249   if List.length list = 0
250     then false
251     else match (List.hd (List.rev list)) with
252       Return(_) -> true
253     | _ -> false
254
255
256 (* let if_else_has_return_stmt stmt_list =
257   let if_stmts = List.filter (function If(_,_,_) -> true | _ -> false)
        stmt_list in
258     let rets = List.map (
259       function
260         If(_,s1,s2) ->
261           begin
262             match s1,s2 with
263               Block(lst1),Block(lst2) -> (has_return_stmt lst1) && (
                    has_return_stmt lst2)
264             | _ -> raise(Failure("Error"))
265           end
266       | _  -> false
```

```
267     ) if_stmts in
268       List.fold_left (fun b v -> b || v) false rets *)
269
270 let has_return_stmt func =
271   let stmt_list = func.body in
272     if List.length stmt_list = 0
273       then false
274       else match List.hd (List.rev stmt_list), func.fname with
275         Return(e),"main" -> raise(Failure("Return statement not permitted in
              main method"))
276       | _, "main" -> false
277       | Return(e), _ -> true
278       | _,_ -> false
279
280
281 (*Returns the type of a given variable name *)
282 let get_type func name =
283   if exists_variable_decl func name (* True if there exists a var of that
        name *)
284     then get_var_type func name
285     else
286       if exists_formal_param func name
287         then get_fparam_type func name
288         else (*Variable has not been declared as it was not found*)
289           let e = "Variable \"" ^ name ^ "\" is being used without being
                declared in function \"" ^ func.fname ^ "\"" in
290             raise (Failure e)
291
292
293 (* Check that the body is valid *)
294 let valid_body func env =
295   (* Check all statements in a block recursively, will throw error for an
        invalid stmt *)
296   let rec check_stmt = function
297       Block(stmt_list) -> let _ = List.map(fun s -> check_stmt s) stmt_list
              in
298         true
299     | Expr(expr) -> let _ = valid_expr func expr env in
300       true
301     | Return(expr) -> let _ = valid_expr func expr env in
302       true
303     | If(condition, then_stmts, else_stmts) -> let cond_type = get_expr_type
          condition func in
304       begin
305         match cond_type with
306             BooleanType ->
307               if (check_stmt then_stmts) && (check_stmt else_stmts)
308                 then true
309                 else raise( Failure("Invalid statements in If statement
                      within function \"" ^ func.fname ^ "\""))
```

```
310           | _ -> raise( Failure("Condition of If statement is not a valid
                boolean expression within function \"" ^ func.fname ^ "\"") )
311       end
312     | For(init, condition, do_expr, stmts) -> let cond_type = get_expr_type
          condition func in
313       let _ = valid_expr func do_expr env in
314         let _ = valid_expr func init env in
315           begin
316             match cond_type with
317                 BooleanType ->
318                   if check_stmt stmts
319                     then true
320                     else raise( Failure("Invalid statements in For loop
                          within function \"" ^ func.fname ^ "\""))
321               | _ -> raise( Failure("Condition of For loop is not a valid
                    boolean expression within function \"" ^ func.fname ^ "\"")
                     )
322           end
323     | While(condition, stmts) -> let cond_type = get_expr_type condition func
          in
324       begin
325         match cond_type with
326             BooleanType ->
327               if check_stmt stmts
328                 then true
329               else raise( Failure("Invalid statments in While loop within
                    function \"" ^ func.fname ^ "\"") )
330           | _ -> raise( Failure("Condition of While loop is not a valid
                boolean expression within function \"" ^ func.fname ^ "\"") )
331       end
332     | Print(expr) -> let expr_type = get_expr_type expr func in
333       begin
334         match expr_type with
335             StringType -> true
336           | _ -> raise( Failure("Print in function \"" ^ func.fname ^ "\"
                does not match string type") )
337       end
338   in
339     let _ = List.map(fun s -> check_stmt s) func.body in
340       true
341
342 let valid_func env f =
343   let duplicate_functions = function_exist f env in
344     (* let duplicate_parameters = count_function_params f in *)
345       let v_body = valid_body f env in
346         let _ = env.functions <- f :: env.functions (* Adding function to
                environment *) in
347         (not duplicate_functions) && (* (not duplicate_parameters) && *)
              v_body
348
```

```
let check_program flist =
  let (environment : env) = { functions = [] (* ; variables = [] *) } in
    let _validate = List.map ( fun f -> valid_func environment f) flist in
      let _ = print_endline "\nSemantic analysis completed successfully.\
          nCompiling...\n" in
        true
```

../compile.ml

```
open Ast
open Str
open Printf
open Parser
module StringMap = Map.Make(String);;

let string_of_type = function
      IntType -> "int"
    | BooleanType -> "Boolean"
    | StringType -> "String"
    | DoubleType -> "double"
    | _ -> ""

let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mul -> "*"
  | Div -> "/"
  | Gt -> ">"
  | Geq -> ">="
  | Lt -> "<"
  | Leq -> "<="
  | Equal -> "=="
  | Neq -> "!="

let string_of_re = function
  And -> "&&"
  | Or -> "||"

let string_of_boolean = function
  True -> string_of_bool true
  | False -> string_of_bool false

let string_of_var = function
 Var(v)-> v


let string_of_rule = function
      Balance(equation) -> "Balance(" ^  equation ^ ");"
    | Mass(equation)-> "Mass(" ^ equation ^ ");"

```

```
42 let rec string_of_expr = function
43   Int(i) -> string_of_int i
44   | Double(d) -> string_of_float d
45   | Boolean(b) -> string_of_boolean b
46   | String (s) -> s
47   | Asn(id, left) -> id ^ " = " ^ (string_of_expr left)
48   | Seq(s1, s2) -> (string_of_expr s1) ^ " ; " ^ (string_of_expr s2)
49   | Call(s,l) -> s ^ "(" ^ String.concat "" (List.map string_of_expr l) ^ ")"
50   | Access(o,m) -> (string_of_expr o) ^ "." ^ m ^"();"
51   | Draw(s, e1, e2, e3, e4, e5, e6, e7, e8) -> "randx = (int) (Math.random()
       *400); randy = (int) (Math.random()*400); scene.add(new AtomShape(randx
       , randy," ^ s ^ "," ^
52     (string_of_int e1) ^ "," ^
53     (string_of_int e2) ^ "," ^
54     (string_of_int e3) ^ "," ^
55     (string_of_int e4) ^ "," ^
56     (string_of_int e5) ^ "," ^
57     (string_of_int e6) ^ "," ^
58     (string_of_int e7) ^ "," ^
59     (string_of_int e8) ^ "))";
60   | Binop (e1, op, e2) ->
61   (string_of_expr e1) ^ " " ^ (match op with
62     Add -> "+"
63     | Sub -> "-"
64     | Mul -> "*"
65     | Div -> "/"
66     | Gt -> ">"
67     | Geq -> ">="
68     | Lt -> "<"
69     | Leq -> "<="
70     | Equal -> "=="
71     | Neq -> "!=")
72     ^ " " ^ (string_of_expr e2)
73   | Brela (e1, op, e2) ->
74   (string_of_expr e1) ^ " " ^ (match op with
75         And -> "&&"
76     | Or -> "||")
77     ^ " " ^ (string_of_expr e2)
78     | Noexpr -> ""
79     | Null -> "NULL"
80     | Concat(s1, s2) -> string_of_expr s1 ^ "+" ^ string_of_expr s2
81     | List(elist) -> "[" ^  String.concat ", " (List.map string_of_expr elist
         ) ^ "]"
82       | Print(s) -> "System.out.println(" ^ string_of_expr s ^ ");"
83     | Equation(name, rlist, plist) -> "equation" ^ name ^ "{"  ^ String.
         concat "," (List.map string_of_var rlist) ^ "--" ^ String.concat ","
         (List.map string_of_var plist) ^ "}"
84
85  (*    | Element(name, mass, electron, charge) -> "element " ^ name ^ "(" ^ (
      string_of_int mass) ^ "," ^ (string_of_int electron) ^ "," ^ (
```

```
     string_of_int charge) ^ ")"
86     | Molecule(name ,elist) -> "molecule " ^ name ^ "{" ^ String.concat "," (
         List.map string_of_var elist) ^ "}" *)
87 let string_of_edecl edecl = "Element " ^ edecl.name ^ "= new Element(" ^ (
     string_of_int edecl.mass) ^ "," ^ (string_of_int edecl.electrons) ^ "," ^
     (string_of_int edecl.charge) ^ ");"
88 let string_of_mdecl mdecl =  "ArrayList<Element> " ^ mdecl.mname ^ "1 = new
     ArrayList<Element>(Arrays.asList(" ^ String.concat "," (List.map
     string_of_var mdecl.elements) ^ "));" ^
89 "Molecule " ^ mdecl.mname ^ "= new Molecule("^ mdecl.mname ^ "1);"
90
91 let string_of_pdecl pdecl = string_of_type pdecl.paramtype ^ " " ^ pdecl.
     paramname
92 let string_of_pdecl_list pdecl_list = String.concat "" (List.map
     string_of_pdecl pdecl_list)
93 let string_of_vdecl vdecl = string_of_type vdecl.vtype ^ " " ^ vdecl.vname ^
     ";\n"
94
95 let rec string_of_stmt = function
96       Block(stmts) ->
97         "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
98     | Expr(expr) -> string_of_expr expr ^ ";\n"
99     | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n"
100    | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n{" ^ (
         string_of_stmt s) ^ "}"
101    | If(e, s1, s2) -> "if (" ^ string_of_expr e ^ ")\n{" ^ (string_of_stmt
         s1) ^ "}\n" ^ "else\n{" ^ (string_of_stmt s2) ^ "}"
102    | For(e1, e2, e3, s) ->
103      "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
104      string_of_expr e3  ^ ") " ^ string_of_stmt s
105    | While(e, s) -> "while (" ^ string_of_expr e ^ ") {" ^ (string_of_stmt s
         ) ^ "}"
106    | Print(s) -> "System.out.println(" ^ string_of_expr s ^ ");"
107
108
109
110 let string_of_vdecl vdecl=
111     string_of_type vdecl.vtype ^ " " ^ vdecl.vname ^ ";"
112
113 let string_of_fdecl fdecl =
114     if fdecl.fname = "main" then "public static void main(String args[])\n{\n
         " ^
115   String.concat "" (List.map string_of_vdecl fdecl.locals) ^
116   String.concat "" (List.map string_of_edecl fdecl.elements) ^
117   String.concat "" (List.map string_of_mdecl fdecl.molecules) ^
118   String.concat "" (List.map string_of_rule fdecl.rules) ^
119   String.concat "" (List.map string_of_stmt fdecl.body) ^
120   "}\n"
121 else
```

```
122   "public static void " ^ fdecl.fname ^ "(" ^ String.concat ", " (List.map
          string_of_pdecl fdecl.formals) ^ ")\n{\n" ^
123   String.concat "" (List.map string_of_vdecl fdecl.locals) ^
124   String.concat "" (List.map string_of_edecl fdecl.elements) ^
125   String.concat "" (List.map string_of_mdecl fdecl.molecules) ^
126   String.concat "" (List.map string_of_rule fdecl.rules) ^
127   String.concat "" (List.map string_of_stmt fdecl.body) ^
128   "}\n"
129
130 let string_of_fdecl_list fdecl_list =
131     String.concat "" (List.map string_of_fdecl fdecl_list)
132
133 let string_of_program (vars, funcs) =
134   String.concat "" (List.map string_of_vdecl (List.rev vars) ) ^ "\n" ^
135   String.concat "\n" (List.map string_of_fdecl (List.rev funcs) ) ^ "\n"
136
137
138  let rec mass_sum element_list = match element_list with
139   | [] -> 0
140   | hd :: tl -> hd.mass + mass_sum tl;;
141
142
143 let rec charge_sum molecule = match molecule with
144   | [] -> 0
145   | hd :: tl -> hd.charge + charge_sum tl;;
146
147
148 let contains s1 s2 =
149     let re = Str.regexp_string s2
150     in
151         try ignore (Str.search_forward re s1 0); true
152         with Not_found -> false
153
154
155
156 let program program prog_name =
157     let jframe a b =
158     if contains (string_of_fdecl_list program) "graphics" then a else b in
159   let out_chan = open_out ("ChemLAB" ^ ".java") in
160     ignore(Printf.fprintf out_chan
161 "
162 import com.graphics.*;
163 import java.util.*;
164 import java.awt.*;
165 import java.awt.event.*;
166 import java.util.ArrayList;
167 import javax.swing.*;
168
169 public class ChemLAB %s
170 {
```

```
171     %s
172     public static boolean debug = false;
173     public static int randx;
174     public static int randy;
175
176     public ChemLAB()
177     {
178         %s
179     }
180
181     public static void Balance(String s)
182     {
183         String[] r = s.split(\"(, )|(==)|(' ')\");
184         String[] r1 = s.split(\"\\\\s*(,|\\\\s)\\\\s*\");
185         String[] r2 = s.split(\"(, )|(' ')\");
186         String[] individual = s.split(\"(, )|(== )|(?=\\\\p{Upper})|(' ')\");
187
188         ArrayList<String> elements = new ArrayList<String>();
189
190         int counter = 0;
191         for(int i=0; i<r2.length; i++){
192             if(r2[i].contains(\"=\"))
193                 counter = i;
194         }
195         counter++;
196
197         for (int i = 0; i < individual.length; i++) {
198             String x = \"\";
199             for (int j = 0; j < individual[i].length(); j++) {
200                 if (Character.isLetter(individual[i].charAt(j)))
201                     x = x + individual[i].charAt(j);
202             }
203             if (!elements.contains(x) && (x != \"\"))
204                 elements.add(x);
205         }
206
207         double[][] matrix = new double [elements.size()][r.length];
208
209         for (int i = 0; i < elements.size(); i++) {
210             String temp = elements.get(i);
211             for (int j = 0; j < r.length; j++) {
212                 if (r[j].contains(temp)) {
213                     int k = r[j].indexOf(temp) + temp.length();
214                     if (k >= r[j].length()) {
215                         k = 0;
216                     }
217                     if (Character.isDigit(r[j].charAt(k))) {
218                         int dig = Integer.parseInt(r[j].substring(k, k + 1));
219                         matrix[i][j] = dig;
220                     } else {
```

```
                    matrix[i][j] = 1;
                }
            } else {
                matrix[i][j] = 0;
            }
        }
    }



    double[][] A = new double[matrix.length][matrix[0].length - 1];
    double[][] B = new double[matrix.length][1];

    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < matrix[i].length - 1; j++) {
            A[i][j] = matrix[i][j];
        }
    }

    int n = A[0].length<A.length? A.length : A[0].length;
    int difference = Math.abs(A.length-A[0].length);
    double[][] A1 = new double[n][n];

    for (int i = 0; i < B.length; i++) {
        B[i][0] = matrix[i][matrix[i].length - 1];
    }


    for(int i = 0; i < A.length; i++)
    {
        for(int j = 0; j < A[0].length; j++)
        {
            A1[i][j] = A[i][j];
        }
    }

    if(A[0].length<A.length){
        for(int i=0; i<n; i++){
            for(int j = n-difference; j< n; j++)
            {
                A1[i][j] = 1;
            }
        }
    }
    else if (A[0].length> A.length)
    {
        for(int i=0; i<n; i++){
            for(int j = n-difference; j< n; j++)
            {
                A1[j][i] = 1;
```

```
271                 }
272             }
273         }
274 
275         for(int i=0; i<n; i++)
276         {
277             for(int j=counter; j<n; j++){
278                 matrix[i][j] = matrix[i][j] * -1;
279             }
280         }
281 
282         double det = determinant(A1, n);
283         double inverse[][] = invert(A1);
284         double[][] prod = product(inverse, B, det);
285 
286         double factor = 0;
287         boolean simplified = true;
288         for(int i = 0; i < prod.length; i++)
289         {
290             for(int j = i; j < prod.length; j++)
291             {
292                 if(mod(prod[i][0],prod[j][0]))
293                 {
294                     simplified = false;
295                     break;
296                 }
297             }
298         }
299 
300         if (simplified == false)
301         {
302             factor = findSmallest(prod);
303             simplify(prod, factor);
304 
305         }
306         boolean subtract = false;
307 
308         for(int j = 0; j < r1.length; j++)
309         {
310             if(j == r1.length-1)
311             {
312                 int sum = 0;
313                 int count = 0;
314                 for(int m = 0; m < B[0].length; m++)
315                 {
316                     if(B[m][0] == 0)
317                     {
318                         count++;
319                     }
320                 }
```

```
321                 for(int k = 0; k < n; k++)
322                 {
323                     sum += Math.round(matrix[count][k]*Math.abs(prod[k][0]));
324 
325                 }
326 
327                 if(B[count][0] == 0)
328                 {
329 
330                     System.out.println(1 + \" \" + r2[j-2]);
331                 }
332                 else
333                 {
334 
335                     System.out.println(Math.abs(sum/(int)B[count][0]) + \" \"
                            + r2[j-2]);
336                 }
337             }
338             else if(r1[j].equals(\"==\"))
339             {
340                 System.out.print(\"--> \");
341                 subtract = true;
342             }
343             else if (subtract == true)
344             {
345                 int coeff = (int)Math.round(Math.abs(prod[j-1][0]));
346                 System.out.print(coeff + \" \" + r1[j] + \" \");
347             }
348             else
349             {
350                 int coeff = (int)Math.round(Math.abs(prod[j][0]));
351                 System.out.print(coeff + \" \" + r1[j] + \" \");
352             }
353         }
354 
355     }
356 
357     public static boolean mod(double a, double b)
358     {
359 
360       int c = (int)(a)/(int)(b);
361       if (c*b == a)
362         return true;
363       else
364         return false;
365     }
366 
367     public static void printMatrix(double[][] matrix)
368     {
369         for (int i = 0; i < matrix.length; i ++)
```

```
370         {
371             for(int j = 0; j< matrix[0].length; j++)
372             {
373                 System.out.print(matrix[i][j] + \" \");
374             }
375             System.out.print(\"\\n\");
376         }
377     }
378 
379     public static double findSmallest(double a[][])
380     {
381         double smallest = a[0][0];
382         for(int i = 0; i < a.length; i++)
383         {
384             if(Math.abs(a[i][0]) < Math.abs(smallest))
385                 smallest = a[i][0];
386         }
387         return smallest;
388     }
389 
390     public static double[][] simplify(double a[][], double smallest)
391     {
392         int largest = 0;
393         boolean all = true;
394         for(int i = 1; i <=  Math.abs(smallest); i++)
395         {
396             all = true;
397             for(int j = 0; j < a.length; j++)
398             {
399                 if(!mod(a[j][0],i) )
400                 {
401                     all = false;
402                 }
403             }
404             if (Math.abs(i)>Math.abs(largest) && all == true)
405                 largest = i;
406         }
407         if (debug == true)
408             System.out.println(largest);
409         if(largest!=0)
410         {
411             for(int k = 0; k < a.length; k++)
412             {
413                 a[k][0] = a[k][0]/largest;
414             }
415         }
416         return a;
417     }
418 
419     public static double[][] product(double a[][], double b[][], double det)
```

```
420     {
421         int rowsInA = a.length;
422        int columnsInA = a[0].length; // same as rows in B
423        int columnsInB = b[0].length;
424        double[][] c = new double[rowsInA][columnsInB];
425        for (int i = 0; i < rowsInA; i++) {
426          for (int j = 0; j < columnsInB; j++) {
427              for (int k = 0; k < columnsInA; k++) {
428                  c[i][j] = c[i][j] + a[i][k] * b[k][j];
429              }
430          }
431      }
432
433      for(int i = 0; i < rowsInA; i++)
434      {
435         c[i][0] = c[i][0]*det;
436     }
437     return c;
438 }
439 public static double determinant(double A[][],int N)
440 {
441     double det=0;
442     if(N == 1)
443     {
444         det = A[0][0];
445     }
446     else if (N == 2)
447     {
448         det = A[0][0]*A[1][1] - A[1][0]*A[0][1];
449     }
450     else
451     {
452         det=0;
453         for(int j1=0;j1<N;j1++)
454         {
455             double[][] m = new double[N-1][];
456             for(int k=0;k<(N-1);k++)
457             {
458                 m[k] = new double[N-1];
459             }
460             for(int i=1;i<N;i++)
461             {
462                 int j2=0;
463                 for(int j=0;j<N;j++)
464                 {
465                     if(j == j1)
466                         continue;
467                    m[i-1][j2] = A[i][j];
468                     j2++;
469                 }
```

```
470             }
471             det += Math.pow(-1.0,1.0+j1+1.0)* A[0][j1] * determinant(m,N-1);
472         }
473     }
474     return det;
475 }
476 public static double[][] invert(double a[][])
477 {
478     int n = a.length;
479     double x[][] = new double[n][n];
480     double b[][] = new double[n][n];
481     int index[] = new int[n];
482     for (int i=0; i<n; ++i)
483         b[i][i] = 1;
484 
485     gaussian(a, index);
486 
487     for (int i=0; i<n-1; ++i)
488         for (int j=i+1; j<n; ++j)
489             for (int k=0; k<n; ++k)
490                 b[index[j]][k]
491             -= a[index[j]][i]*b[index[i]][k];
492 
493             for (int i=0; i<n; ++i)
494             {
495                 x[n-1][i] = b[index[n-1]][i]/a[index[n-1]][n-1];
496                 for (int j=n-2; j>=0; --j)
497                 {
498                     x[j][i] = b[index[j]][i];
499                     for (int k=j+1; k<n; ++k)
500                     {
501                         x[j][i] -= a[index[j]][k]*x[k][i];
502                     }
503                     x[j][i] /= a[index[j]][j];
504                 }
505             }
506             return x;
507         }
508 
509 // Method to carry out the partial-pivoting Gaussian
510 // elimination.  Here index[] stores pivoting order.
511 
512         public static void gaussian(double a[][], int index[])
513         {
514             int n = index.length;
515             double c[] = new double[n];
516 
517  // Initialize the index
518             for (int i=0; i<n; ++i)
519                 index[i] = i;
```

```
520
521 // Find the rescaling factors, one from each row
522             for (int i=0; i<n; ++i)
523             {
524                 double c1 = 0;
525                 for (int j=0; j<n; ++j)
526                 {
527                     double c0 = Math.abs(a[i][j]);
528                     if (c0 > c1) c1 = c0;
529                 }
530                 c[i] = c1;
531             }
532
533 // Search the pivoting element from each column
534             int k = 0;
535             for (int j=0; j<n-1; ++j)
536             {
537                 double pi1 = 0;
538                 for (int i=j; i<n; ++i)
539                 {
540                     double pi0 = Math.abs(a[index[i]][j]);
541                     pi0 /= c[index[i]];
542                     if (pi0 > pi1)
543                     {
544                         pi1 = pi0;
545                         k = i;
546                     }
547                 }
548
549   // Interchange rows according to the pivoting order
550                 int itmp = index[j];
551                 index[j] = index[k];
552                 index[k] = itmp;
553                 for (int i=j+1; i<n; ++i)
554                 {
555                     double pj = a[index[i]][j]/a[index[j]][j];
556
557 // Record pivoting ratios below the diagonal
558                     a[index[i]][j] = pj;
559
560 // Modify other elements accordingly
561                     for (int l=j+1; l<n; ++l)
562                         a[index[i]][l] -= pj*a[index[j]][l];
563                 }
564             }
565         }
566         %s
567     }"  (jframe "extends JFrame" "") (jframe "final static SceneComponent
        scene = new SceneComponent();" "")
```

```
568     (jframe "setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); setSize(500,
            500); add(scene, BorderLayout.CENTER);" "") (string_of_fdecl_list
            program ) );
569         close_out out_chan;
570         ignore(Sys.command ("javac ChemLAB.java"));
571         ignore(Sys.command (Printf.sprintf "java %s" "ChemLAB"));
572     let contains s1 s2 =
573     let re = Str.regexp_string s2
574     in
575         try ignore (Str.search_forward re s1 0); true
576         with Not_found -> false
577     in
578                 if (contains (string_of_fdecl_list program) "graphics") then
                        (ignore(Sys.command ("javac ChemLAB.java SceneEditor.java
                        ")); ignore(Sys.command("java SceneEditor")));
```

../chemlab.ml

```
1  exception NoInputFile
2  exception InvalidProgram
3
4  let usage = Printf.sprintf "Usage: chemlab FILE_NAME"
5  (* Get the name of the program from the file name. *)
6  let get_prog_name source_file_path =
7    let split_path = (Str.split (Str.regexp_string "/") source_file_path) in
8    let file_name = List.nth split_path ((List.length split_path) - 1) in
9    let split_name = (Str.split (Str.regexp_string ".") file_name) in
10     List.nth split_name ((List.length split_name) - 2)
11
12 (* Entry Point: starts here *)
13 let _ =
14   try
15     let prog_name =
16       if Array.length Sys.argv > 1 then
17         get_prog_name Sys.argv.(1)
18       else raise NoInputFile in
19
20     let input_channel = open_in Sys.argv.(1) in
21
22     let lexbuf = Lexing.from_channel input_channel in
23       let prog = Parser.program Scanner.token lexbuf in
24         (* if Semantic.check_program prog
25           then *) Compile.program prog prog_name
26           (* else raise InvalidProgram *)
27   with (* Not sure why this wants an int instead of unit *)
28     | NoInputFile -> ignore(Printf.printf "Please provide a name for a
          ChemLAB file\n");print_endline ""
29     | InvalidProgram -> ignore(Printf.printf "Invalid program\n");
          print_endline ""
```

../test.sh

```
#!/bin/bash

make
#javac ChemLAB.java
#java ChemLAB

FILES="test/*.chem"

#for f in $FILES
#do
# testname=${f%.chem}
# echo
# echo -ne "##### Testing " #-ne means no new line
# echo $testname
# ./chemlab $f
#done
#./chemlab test/test1.chem
./chemlab test/test9.chem
./chemlab test/test1.chem
# echo
# echo "Cleaning up..."
# make clean
```