

ChemLAB
COMS W4115 - Programming Languages & Translators
Professor Stephen Edwards

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

December 16, 2014

Contents

Introduction	2
1 Language Tutorial	3
1.1 Program Execution	3
1.2 Variables	3
1.3 Control Flow	3
1.4 Functions	4
1.5 Printing to stdout	4
2 Language Reference Manual	5
2.1 Types	5
2.1.1 Primitive Types	5
2.1.2 Non-Primitive Types	6
2.1.3 Type Inference	7
2.2 Lexical Conventions	7
2.2.1 Identifiers	7
2.2.2 Keywords	7
2.2.3 Literals	7
2.2.4 Punctuation	8
2.2.5 Comments	8
2.2.6 Operators	9
2.3 Syntax	9
2.3.1 Expressions	10
2.3.2 Statements	13
2.3.3 Scope	14
2.4 Built-in Functions	15
3 Project Plan	16
3.1 Proposed Plan	16
3.2 What Actually Happened	17

3.3	Team Responsibilities	17
3.4	Project Log	18
4	Architectural Design	19
4.1	Scanning	19
4.2	Parsing and Abstract Syntax Tree	19
4.3	Semantic Analysis	20
4.4	Java Generation	20
5	Test Plan	21
5.1	Introduction	21
5.2	Test Cases	21
6	Lessons Learned	25
A	Code Listing	26

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, stoichiometric 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

`make` creates an executable `chemlab`. To compile and run a `.chem` program, simply run the executable `chemlab` with your `.chem` file as the only argument. `./chemlab <program name>.chem`

It then compiles the ChemLab file into Java bytecode, which is then executed on a Java virtual machine.

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. All programs in ChemLAB must contain one “main” function which is the starting point for the program. 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 main(int A, int B){
    print A;
}
```

This is a function that takes in no parameters:

```
function main(){
print "Hello World";
}
```

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

2.1 Types

2.1.1 Primitive Types

There are four primitive types in ChemLab: boolean, int, double, and string.

Boolean

The boolean data type has only two possible values: true and false. The boolean data type can be manipulated in boolean expressions involving the AND, OR, and NOT operators.

Integers

Much like in the Java programming language, the int data type is represented with 32-bits and in signed two's complement form. It has a minimum value of -2^{31} and maximum value of 2^{31} . There is no automatic type conversion between a variable of type int and of type double. In fact, an error will occur when the two primitive types are intermixed.

Double

Much like in the Java programming language, a double is a double-precision 64-bit IEEE 754 floating point with values ranging from $4.94065645841246544e - 324d$ to $1.79769313486231570e + 308d$ (positive or negative). Double should be used under any circumstance when there are decimal values.

String

Unlike in the C programming language, a string is a primitive type rather than a collection of characters. A string is a sequence of characters surrounded by double

quotes “”. Our language supports string concatenation. In the context of strings, the “+” operator concatenates two strings together to form a new string.

2.1.2 Non-Primitive Types

The language comes built-in with lists, elements, molecules, equation.

Lists

A list is a collection of items that maintains the order in which the items were added much like an ArrayList in Java. The type of items in a list must be declared and the type must remain consistent throughout the lifetime of the program. A list is declared in a syntax very similar to declaration in Java:

```
<type> <identifier>[] = [ element_1, element_2, ....., element_n]
```

Element

Since there are only 118 elements, it could have been possible to hard code each element into the language. However, we chose not to do this to give the user a greater degree of flexibility in terms of declaring the properties of the element they want to consider because isotopes of elements have different amounts of neutrons and some elements can exist in more than one state. Element is declared with (atomic number, mass number, charge). The element type is the basic building block provided by the program that can be used to create molecules, compounds, etc. Elements are immutable.

$^{12}_6C$ is represented as: `element C(6, 12, 0);`

$^{14}_6C$ is represented as: `element C(6, 14, 0);`

Molecule

For the purpose of the language, there is no distinction between molecule or compound and both are declared the same way. A molecule is declared as a list of elements surrounded by braces.

NaCl is represented as: `molecule NaCl {[Na, Cl]}`

Equation

Equation is declared in the following way: (list of elements/molecules on left side of reaction, list of elements/molecules on right side of reaction). Underneath, it is essentially, two lists that keep track of the two sides of the equation.

`<equationName>.right` or `<equationName>.left` allows easy access to one side of the equation. Once declared, an equation is immutable.

$NaOH + HCl \rightarrow NaCl + H_2O$ is represented as:


```
equation NaClReaction = {[NaOH, HCl], [NaCl, H2O]};
```

2.1.3 Type Inference

The language is not type-inferred, making it necessary to explicitly declare types.

2.2 Lexical Conventions

2.2.1 Identifiers

An identifier is a sequence of letters or digits in which the first character must be a uppercase letter. Our language is case sensitive, so upper and lower case letters are considered different.

2.2.2 Keywords

The following identifiers start with a lowercase letter and are reserved for use as keywords, and may not be used otherwise:

- | | | |
|------------|------------|---------|
| • int | • equation | • true |
| • double | • if | • false |
| • string | • else | • print |
| • boolean | • while | • call |
| • element | • function | |
| • molecule | • return | |

2.2.3 Literals

Literals are values written in conventional form whose value is obvious. Unlike variables, literals do not change in value. An integer or double literal is a sequence of digits. A boolean literal has two possible values: true or false.

2.2.4 Punctuation

These following characters have their own syntactic and semantic significance and are not considered operators or identifiers.

Punctuator	Use	Example
,	List separator, function parameters	<code>function int sum(int a, int b);</code>
;	Statement end	<code>int x = 3;</code>
"	String declaration	<code>string x = "hello";</code>
[]	List delimiter	<code>int x[] = [1, 2, 3];</code>
{}	Statement list delimiting, and element/molecule/equation declaration	<code>if(expr) { statements }</code>
()	Conditional parameter delimiter, expression precedence	<code>while(i > 2)</code>

2.2.5 Comments

Much like in the C programming language, the characters `/*` introduce a comment, which terminates with the characters `*/`. Single line comments start with `//` and end at the new line character `\n`.

2.2.6 Operators

Operator	Use	Associativity
=	Assignment	Right
==	Test equivalence	Left
!=	Test inequality	Left
>	Greater than	Left
<	Less than	Left
>=	Greater than or equal to	Left
<=	Less than or equal to	Left
&&	AND	Left
	OR	Left
.	Access	Left
*	Multiplication	Left
/	Division	Left
+	Addition	Left
-	Subtraction	Left
^	Concatenate	Left
%	Modulo	Left

The precedence of operators is as follows (from highest to lowest):

1. * / %
2. + -
3. < > <= >=
4. == !=
5. &&
6. ||
7. .
8. ^
9. =

2.3 Syntax

A program in ChemLab consists of at least one function, where one of them is named “main”. Within each function there is a sequence of zero or more valid ChemLab state-

ments.

2.3.1 Expressions

An expression is a sequence of operators and operands that produce a value. Expressions have a type and a value and the operands of expressions must have compatible types. The order of evaluation of subexpressions depends on the precedence of the operators but, the subexpressions themselves are evaluated from left to right.

Constants

Constants can either be of type boolean, string, int, or double.

Identifiers

An identifier can identify a primitive type, non-primitive type, or a function. The type and value of the identifier is determined by its designation. The value of the identifier can change throughout the program, but the value that it can take on is restricted by the type of the identifier. Furthermore, after an identifier is declared, there can be no other identifiers of the same name declared within the scope of the whole program.

```
int x = 3;
x = true; //syntax error
boolean x = 5; //error, x has already been declared
```

Binary Operators

Binary operators can be used in combination with variables and constants in order to create complex expressions. A binary operator is of the form : <expression> <binary-operator> <expression>

Arithmetic operators Arithmetic operators include *, /, %, +, and -. The operands to an arithmetic operator must be numbers. the type of an arithmetic operator expression is either an int or a double and the value is the result of calculating the expression. Note, can not do arithmetic operations when the values involved are a mix of int and double.

`expression * expression`

The binary operator * indicates multiplication. It must be performed between two int types or two double types. No other combinations are allowed.

expression / expression

The binary operator / indicates division. The same type considerations as for multiplication apply.

expression % expression

The binary operator % returns the remainder when the first expression is divided by the second expression. Modulo is only defined for int values that have a positive value.

expression + expression

The binary operator + indicates addition and returns the sum of the two expressions. The same type considerations as for multiplication apply.

expression - expression

The binary operator - indicates subtraction and returns the difference of the two expressions. The same type considerations as for multiplication apply.

Relational operators Relational operators include <, >, <=, >=, ==, and !=. The type of a relational operator expression is a boolean and the value is true if the relation is true while it is false if the relation is false.

expression1 > expression2

The overall expression returns true if expression1 is greater than expression 2

expression1 < expression2

The overall expression returns true if expression1 is less than expression 2

expression1 >= expression2

The overall expression returns true if expression1 is greater than or equal to expression 2

expression1 <= expression2

The overall expression returns true if expression1 is less than or equal to expression 2

expression1 == expression2

The overall expression returns true if expression1 is equal to expression 2.

expression1 != expression2

The overall expression returns true if expression1 is not equal to expression 2

Assignment operator The assignment operator (=) assigns whatever is on the right side of the operator to whatever is on the left side of the operator

expression1 = expression2

expression1 now contains the value of expression2

Access operator The access operator is of the form `expression.value`. The expression returns the value associated with the particular parameter. The expression must be of a non-primitive type.

Logical operators Logical operators include AND (`&&`) and OR (`||`). The operands to a logical operator must both be booleans and the result of the expression is also a boolean.

`expression1 && expression2`

The overall expression returns true if and only if `expression1` evaluates to true and `expression2` also evaluates to true.

`expression1 || expression2`

The overall expression returns true as long as `expression1` and `expression2` both do not evaluate to false.

Parenthesized Expression

Any expression surrounded by parentheses has the same type and value as it would without parentheses. The parentheses merely change the precedence in which operators are performed in the expression.

Function Creation

The syntax for declaration of a function is as follows

```
function functionName (type parameter1, type parameter 2, ...) {  
    statements  
}
```

The function keyword signifies that the expression is a function. Parameter declaration is surrounded by parentheses where the individual parameters are separated by commas. All statements in the function must be contained within the curly braces. A good programming practice in ChemLab is to declare all the functions at the beginning of the program so that the functions will definitely be recognized within the main of the program.

Function Call

Calling a function executes the function and blocks program execution until the function is completed. When a function is called, the types of the parameter passed into the function must be the same as those in the function declaration. The way to call a function is

as follows using the Call keyword: `call functionName(param1, param2, etc...)` When a function with parameters is called, the parameters passed into the function are evaluated from left to right and copied by value into the function's scope. `functionName()` if there are no parameters for the function

2.3.2 Statements

A statement in ChemLab does not produce a value and it does not have a type. An expression is not a valid statement in ChemLab.

Selection Statements

A selection statement executes a set of statements based on the value of a specific expression. In ChemLab, the main type of selection statement is the if-else statement. An if-else statement has the following syntax:

```
if( expression){  
  
}else{  
  
}
```

Expression must evaluate to a value of type boolean. If the expression evaluates to true, then the statements within the first set of curly brackets is evaluated. If the expression evaluates to false, then the statements in the curly brackets following else is evaluated. If-else statements can be embedded within each other. Much like in the C programming language, the dangling if-else problem is resolved by assigning the else to the most recent else-less if. Unlike in Java, an if must be followed by an else. A statement with only if is not syntactically correct.

```
if ( ){  
    if ( ){  
  
    }else{  
  
    }  
}else{  
  
}
```

Iteration Statements

ChemLab does not have a for loop unlike most programming languages. The only iteration statement is the while loop. The while statements evaluates an expression before going into the body of the loop. The expression must be of type boolean and the while loop while continue executing so long as the expression evaluates to true. Once the expression evaluates to false, the while loop terminates. The while loop syntax is as follows:

```
while ( expression ) {  
    statements  
}
```

Note that if values in the expression being evaluated are not altered through each iteration of the loop, there is a risk of going into an infinite loop.

Return Statements

A return statement is specified with the keyword return. In a function, the expression that is returned must be of the type that the function has declared. The syntax of a return statement is: return expression;

The return statement will terminate the function it is embedded in or will end the entire program if it is not contained within a function.

2.3.3 Scope

A block is a set of statements that get enclosed by braces. An identifier appearing within a block is only visible within that block. However, if there are two nested blocks, an identifier is recognizable and can be edited within the nested block.

```
function int notRealMethod(int x){  
    int y = 4;  
    while(x>5){  
        while(z>2){  
            y++;  
        }  
    }  
}
```

In this case, y is recognizable within the second while loop and its value will be incremented. One must also note that, functions only have access to those identifiers that are either declared within their body or are passed in as parameters.

2.4 Built-in Functions

Balance Equations

Given an unbalanced equation, this utility will be able to compute the correct coefficients that go in front of each molecule to make it balanced

Molar Mass Calculation

Given a molecule, this utility will be able to compute the total molar mass of the molecule

Naming of Molecules

Given a molecule, the utility will print out the name in correct scientific notation (ex. H_2O will be printed as Dihydrogen Monoxide)

Printing of Equations

Given an equation, the utility will print out the equation in correct scientific notation

Amount of Moles

Given the element and the amount of grams of the element, this utility will return the amount of moles of the element.

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 OCaml yacc 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

The architectural design of ChemLAB can be divided into the following steps

1. Scanning
2. Parsing
3. Semantic Analysis
4. Java code generation
5. Running the Java code

4.1 Scanning

The ChemLAB scanner tokenizes the input into ChemLAB readable units. This process involves discarding whitespaces and comments. At this stage, illegal character combinations are caught. The scanner was written with `ocamllex`.

4.2 Parsing and Abstract Syntax Tree

The parser generates an abstract syntax tree based on the tokens that were provided by the scanner. Any syntax errors are caught here. The parser was written with `ocamlyacc`.

4.3 Semantic Analysis

The semantic analyzer takes in the AST that was generated by the parser and checks the AST for type errors as well as to make sure that statements and expressions are written in a way that corresponds to the syntax defined by the language. A semantically checked AST (SAST) is not generated. If no errors are thrown, then we can assume that it is safe to use the AST to generate Java code.

4.4 Java Generation

The module walks the AST and generates Java code corresponding to the program. All of the code is put into two Java files. One contains graphics and one contains everything else related to the program. The Java code is generated but not compiled. This needs to be done by the ChemLAB script which will run the javac command.

Chapter 5

Test Plan

5.1 Introduction

To ensure that one person's change and updates would not affect the changes others made previously, an automated test was put in place to run through all the tests to make sure everything that worked before still continued to work. Testing was done using a bash shell script to automate the process. The shell script compiles and runs all the test files and compares them with the expected output. Test cases were written to test individual components of the language such as arithmetic, conditional loops, printing, etc.

5.2 Test Cases

Listing 5.1: Hello World test

```
1 /* Test 1: Hello World (comments, print) */  
2  
3 function main() {  
4     print "Hello , world!";  
5 }
```

Listing 5.2: Int and String Variable Assignment

```
1 /* Test 2: int and string variable assignment */  
2  
3 function main() {  
4     int A;  
5     int B;  
6     string S;
```

```

7
8   A = 2;
9   B = 3;
10  S = "ChemLAB";
11
12  print A;
13  print B;
14  print S;
15 }

```

Listing 5.3: Arithmetic test

```

1  /* Test 3: Arithmetic Expressions */
2
3  function main()
4  {
5      print 0;
6      print 1;
7
8      /* Plus, minus, multiply, divide, mod */
9      print 1+1;
10     print 4-1;
11     print 2*2;
12     print 15/3;
13     print 40%17;
14
15     /* Precedence */
16     print 90-6*8;
17
18     /* Parenthesis */
19     print (1+2*3-4)*28/2;
20
21     /* Negative Numbers */
22     // print -3-39;
23     // print 14*-3;
24
25     /* Decimals */
26     // print 2.1*2;
27     // print 42/99;
28 }

```

Listing 5.4: String Concatenation

```

1  /* Test 4: String Concatenation */
2
3  function main()
4  {
5      string A;
6      string B;
7      string C;

```



```

8 | A = "Hello";
9 | B = "world";
10 | C = "!";
11 | print A ^ ", " ^ B ^ C;
12 | }

```

Listing 5.5: If Condition

```

1 | /* Test 5: If Conditional, Boolean */
2 |
3 | function main() {
4 |     int X;
5 |     int Y;
6 |     X = 17;
7 |     Y = 42;
8 |     if (X < Y) {
9 |         print X + " is less than " + Y;
10 |     } else {
11 |         print "Test Failed";
12 |     }
13 | }

```

Listing 5.6: Nested If Condition

```

1 | /* Test 6: Nested If Else */
2 |
3 | function main()
4 | {
5 |     int X;
6 |     int Y;
7 |     X = 17;
8 |     Y = 39;
9 |
10 |     if (X != Y) {
11 |         Y = Y + 2;
12 |
13 |         if (X > Y) {
14 |             print "Inner If Failed";
15 |         } else {
16 |             Y = Y + 1;
17 |         }
18 |     } else {
19 |         print "Outer If Failed";
20 |     }
21 |
22 |     print Y;
23 | }

```

Listing 5.7: While Loop

```

1  /* Test 8: While Loop */
2
3  function main()
4  {
5      int A;
6      int B;
7      A = 0;
8      B = 3;
9
10
11     while(A < B)
12     {
13         A = A + 1;
14         print A;
15     }
16 }

```

Listing 5.8: Draw

```

1  function main()
2  {
3      int A;
4      element C(12,13,14);
5      A = C.mass;
6      print A;
7
8  }
9  function graphics()
10 {
11     draw("C", 1,1,1,1,0,0,0,0);
12     draw("Na", 0,0,0,0,1,1,0,0);
13     draw("Ne", 1,1,1,1,1,1,1,1);
14     draw("H", 1,0,0,0,0,0,0,0);
15 }

```

Listing 5.9: Balance

```

1  function main ()
2  {
3      element C(12,12,12);
4      balance("HNO3, Cu == CuN2O6, H2O, NO");
5  }

```

Chapter 6

Lessons Learned

Appendix A

Code Listing

Listing A.1: Abstract Syntax Tree (`ast.ml`)

```
1 type operator = Add | Sub | Mul | Div | Mod | Eq | 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   | Bracket of expr
26   | Null
27   | Noexpr
28
29 type stmt =
30   Block of stmt list
```

```

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

```

Listing A.2: Scanner (`scanner.mll`)

```

1 { open Parser }
2
3 let digit = ['0'-'9']
4 let letter = ['A'-'Z' 'a'-'z']
5 let element = ['A'-'Z']['a'-'z']?   (* Symbol of element such as: H, Cl *)
6
7 rule token = parse
8   [ ' ' '\t' '\r' '\n' ]           { token lexbuf }
9   | "/"*                           { comment lexbuf }
10  | "//"                             { line_comment lexbuf }
11  | "("                               { LPAREN }
12  | ")"                               { RPAREN }
13  | "["                               { LBRACKET }
14  | "]"                               { RBRACKET }
15  | "{"                               { LCURLY }
16  | "}"                               { RCURLY }
17  | "\""                             { STRINGDECL }
18  | ";"                               { SEMI }
19  | ":"                               { COLON }
20  | ","                               { COMMA }
21  | "."                               { ACCESS }
22  | "+"                               { PLUS }
23  | "-"                               { MINUS }
24  | "*"                               { TIMES }
25  | "/"                               { DIVIDE }
26  | "%"                               { MOD }
27  | "="                               { ASSIGN }
28  | "^"                               { CONCAT }
29  | "=="                             { EQ }
30  | "!="                             { NEQ }
31  | "<"                               { LT }
32  | "<="                             { LEQ }
33  | ">"                               { GT }
34  | ">="                             { GEQ }
35  | "&&"                             { AND }
36  | "||"                             { OR }
37  | "!"                               { NOT }
38  | "—>"                             { ARROW }
39  | "if"                             { IF }
40  | "else"                           { ELSE }
41  | "while"                           { WHILE }
42  | "for"                             { FOR }
43  | "int"                             { INT }
44  | "double"                          { DOUBLE }
45  | "string"                          { STRING }
46  | "boolean"                         { BOOLEAN }
47  | "element"                         { ELEMENT }
48  | "molecule"                       { MOLECULE }
49  | "equation"                        { EQUATION }

```

```

50 | "balance"          { BALANCE }
51 | "mass"      as attr { ATTRIBUTE(attr) }
52 | "charge"    as attr { ATTRIBUTE(attr) }
53 | "electrons" as attr { ATTRIBUTE(attr) }
54 | "function"  { FUNCTION }
55 | "object"    { OBJECT }
56 | "return"    { RETURN }
57 | "print"     { PRINT }
58 | "call"      { CALL }
59 | "draw"      { DRAW }
60 | "true"      { BOOLEAN_LIT(true) }
61 | "false"     { BOOLEAN_LIT(false) }
62 | digit+ as lxm { INT_LIT(int_of_string lxm) }
63 (* | ([ ' - ' ' + ' ] ) ? ( digit ) * ( ' . ' ) ? digit + ([ ' e ' ' E ' ] [ ' - ' ' + ' ] ? [ ' 0 ' - ' 9 ' ] + ) ? as
   lxm { DOUBLE_LIT(float_of_string lxm) } *)
64 | ( ' 0 ' | [ ' 1 ' - ' 9 ' ] + [ ' 0 ' - ' 9 ' ] * ) ([ ' . ' ] [ ' 0 ' - ' 9 ' ] + ) ? as lxm { DOUBLE_LIT(
   float_of_string lxm) }
65 | ( letter | digit | ' - ' ) * as lxm { ID(lxm) }
66 | ' ' ' [ ^ ' " ' ] * ' ' ' as lxm { STRING_LIT(lxm) }
67 | element as lxm { ELEMENT_LIT(lxm) }
68 | ( element [ ' 0 ' - ' 9 ' ] * ) + as lxm { MOLECULE_LIT(lxm) }
69 | eof { EOF }
70 | - as char { raise (Failure("illegal character " ^
71 | Char.escaped char)) }
72
73 and comment = parse
74 "*/" { token lexbuf }
75 | - { comment lexbuf }
76
77 and line_comment = parse
78 "\n" { token lexbuf }
79 | - { line_comment lexbuf }

```

Listing A.3: Parser (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 %token RETURN IF ELSE FOR WHILE INT DOUBLE STRING BOOLEAN ELEMENT MOLECULE
   EQUATION FUNCTION
11 %token INT DOUBLE STRING BOOLEAN ELEMENT MOLECULE EQUATION FUNCTION
12 %token CALL ACCESS DRAW
13 %token BALANCE MASS CHARGE ELECTRONS

```

```

14 %token AND OR
15 %token INT BOOLEAN STRING DOUBLE
16 %token <string> DATATYPE ATTRIBUTE
17 %token <bool> BOOLEAN_LIT
18 %token <string> ELEMENT_LIT
19 %token <string> MOLECULE_LIT
20 %token <string> STRING_LIT
21 %token <string> ID
22 %token <int> INT_LIT
23 %token <float> DOUBLE_LIT
24 %token EOF
25
26
27 %nonassoc NOELSE
28 %nonassoc ELSE
29 %right ASSIGN
30 %left CONCAT
31 %left ACCESS
32 %left OR
33 %left AND
34 %left EQ NEQ
35 %left LT GT LEQ GEQ
36 %left PLUS MINUS
37 %left TIMES DIVIDE MOD
38 %nonassoc LPAREN RPAREN
39
40 %start program
41 %type <Ast.program> program
42
43 %%
44 program :
45     /* nothing */           { [] }
46     | program fdecl         { ($2 :: $1) }
47
48 id :
49     ID                       { $1 }
50     | STRING_LIT             { $1 }
51     | ELEMENT_LIT            { $1 }
52     | MOLECULE_LIT           { $1 }
53
54 var :
55     id                       { Var($1) }
56
57 vdecl :
58     datatype ID SEMI
59     { { vname = $2;
60       vtype = $1;
61     } }
62
63 vdecl_list :

```



```

64  /* nothing */    {[]}
65  | vdecl_list vdecl {($2::$1)}
66
67  stmt:
68      expr SEMI                { Expr($1) }
69  | RETURN expr SEMI          { Return($2) }
70  | PRINT expr SEMI           { Print($2) }
71  | LCURLY stmt_list RCURLY   { Block(List.rev $2) }
72  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([]) ) }
73  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
74  | FOR LPAREN expr SEMI expr SEMI expr RPAREN stmt { For($3, $5, $7, $9) }
75  | WHILE LPAREN expr RPAREN stmt           { While($3, $5) }
76
77  stmt_list:
78      /* nothing */    { [] }
79  | stmt_list stmt { ($2 :: $1) }
80
81  datatype:
82      INT { IntType }
83  | BOOLEAN { BooleanType }
84  | STRING { StringType }
85  | DOUBLE { DoubleType }
86
87  expr:
88      INT_LIT                { Int($1) }
89  | id                       { String($1) }
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 MOD expr            { Binop($1, Mod, $3) }
96  | expr EQ expr             { Binop($1, Eq, $3) }
97  | expr NEQ expr            { Binop($1, Neq, $3) }
98  | expr LT expr             { Binop($1, Lt, $3) }
99  | expr GT expr             { Binop($1, Gt, $3) }
100 | expr LEQ expr            { Binop($1, Leq, $3) }
101 | expr GEQ expr            { Binop($1, Geq, $3) }
102 | expr AND expr            { Brela($1, And, $3) }
103 | expr OR expr             { Brela($1, Or, $3) }
104 | expr CONCAT expr         { Concat($1, $3) }
105 | id ASSIGN expr           { Asn($1, $3) }
106 | CALL id LPAREN actuals_opt RPAREN { Call($2, $4) }
107 | expr ACCESS ATTRIBUTE    { Access($1, $3) }
108 | 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) }
109 | LPAREN expr RPAREN      { Bracket($2) }
110

```

```

111 edecl:
112     ELEMENT id LPAREN INT_LIT COMMA INT_LIT COMMA INT_LIT RPAREN SEMI
113     {{
114         name = $2;
115         mass = $4;
116         electrons = $6;
117         charge = $8
118     }}
119
120 edecl_list:
121     /* nothing */           { [] }
122     | edecl_list edecl      { List.rev ($2 :: $1)}
123
124
125 mdecl:
126     MOLECULE id LCURLY element_list RCURLY SEMI
127     {{
128         mname = $2;
129         elements = $4;
130     }}
131
132 mdecl_list:
133     /* nothing */           { [] }
134     | mdecl_list mdecl      { ($2 :: $1) }
135
136 element_list:
137     var { [$1] }
138     | element_list COMMA var { ($3 :: $1)}
139
140 rule:
141     BALANCE LPAREN id RPAREN SEMI {Balance($3)}
142
143
144 rule_list:
145     /* nothing */           { [] }
146     | rule_list rule        { ($2 :: $1)}
147
148 formals_opt:
149     /* nothing */           { [] }
150     | formal_list           { List.rev $1 }
151
152 formal_list:
153     param_decl { [$1] }
154     | formal_list COMMA param_decl { $3 :: $1 }
155
156 actuals_opt:
157     /* nothing */           { [] }
158     | actuals_list          { List.rev $1 }
159
160 actuals_list:

```

```

161     expr                { [$1] }
162 | actuals_list COMMA expr { $3 :: $1 }
163
164 param_decl:
165     datatype id
166     { { paramname = $2;
167       paramtype = $1 } }
168
169 fdecl:
170     FUNCTION id LPAREN formals_opt RPAREN LCURLY vdecl_list edecl_list
171           mdecl_list rule_list stmt_list RCURLY
172     { {
173       fname = $2;
174       formals = $4;
175       locals = List.rev $7;
176       elements = List.rev $8;
177       molecules = List.rev $9;
178       rules = List.rev $10;
179       body = List.rev $11
180     } }

```

Listing A.4: Semantic Checker (`semantic.ml`)

```

1  open Ast
2  open Str
3
4  type env = {
5      mutable functions : func_decl list;
6  }
7
8  let function_equal_name name = function
9      func -> func.fname = name
10
11 let function_fparam_name name = function
12     par -> par.paramname = name
13
14 let function_var_name name = function
15     variable -> variable.vname = name
16
17 (* Checks whether a function has been defined duplicately *)
18 let function_exist func env =
19     let name = func.fname in
20     try
21         let _ = List.find (function_equal_name name) env.functions in
22         let e = "Duplicate function: " ^ name ^ " has been defined more than
23             once" in
24             raise (Failure e)
25     with Not_found -> false
26

```

```

27 (*Checks if function has been declared*)
28 let exist_function_name name env = List.exists (function_equal_name name) env
    .functions
29
30
31 let get_function_by_name name env =
32   try
33     let result = List.find (function_equal_name name) env.functions in
34     result
35   with Not_found -> raise(Failure("Function " ^ name ^ " has not been declared
    !"))
36
37
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  exists 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 function*)
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
121     in
122     let e = "Duplicate param: "^ name ^"has been defined more than once" in
123     raise (Failure e)
124   with Not_found -> false
125
126 let get_fparam_type func fparam =
127   try
128     let fparam =
129       *)
130     (*Determines if the given identifier exists*)
131     let exists_id name func = (exists_variable_decl func name) || (
132       exists_formal_param func name)
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 '" ^ mm ^ "' in compute function
171     " ^ func.fname ^ "\n" in
172     let be = e ^ "The only allowed values for initializing boolean
173       variables are 'true' and 'false.' \n" in
174       match vtype with
175       | "Int" -> if is_string value then true else raise (Failure e)
176       | "Double" -> if is_float value then true else raise (Failure e)
177       | "String" -> if is_int value then true else raise (Failure e)
178       | "Boolean" -> if is_string_bool value then true else raise (
179         Failure be)) func.locals
180   in
181   true
182 *)
183
184 let rec get_expr_type e func =
185   match e with
186   | String(s) -> StringType
187   | Int(s) -> IntType
188   | Double(f) -> DoubleType
189   | Boolean(b) -> BooleanType
190   | Binop(e1,op,e2) -> let t1 = get_expr_type e1 func and t2 =
191     get_expr_type e2 func in
192     begin
193       match t1, t2 with
194       | DoubleType, DoubleType -> DoubleType
195       | IntType, IntType -> IntType
196       | _,- -> raise (Failure "Invalid types for binary expression")
197     end
198   | Brela(e1, re, e2) -> let t1 = get_expr_type e1 func and t2 =
199     get_expr_type e2 func in
200     begin
201       match t1, t2 with
202       | BooleanType, BooleanType -> BooleanType
203       | _,- -> raise (Failure "Invalid type for AND, OR expression")
204     end
205   | Asn(expr, expr2) -> get_expr_type expr2 func
206   | Equation (s, vlist, vlist2) -> EquationType
207   | Concat(s, s2) -> let s_type = get_expr_type s func in
208     let s2_type = get_expr_type s2 func in
209     begin
210       match s_type, s2_type with
211       | StringType, StringType -> StringType
212       | _,- -> raise (Failure "concatentation needs to be with two
213         strings")

```

```

209         end
210         | _ -> raise( Failure("!!! Need to implement in get_expr_type: Seq, List,
211                               Call, Null, Noexpr !!!") )
212
213 let rec valid_expr (func : Ast.func_decl) expr env =
214     match expr with
215     | Int(_) -> true
216     | Double(_) -> true
217     | Boolean(_) -> true
218     | String(_) -> true
219     | Binop(e1, _, e2) -> (is_num func e1) && (is_num func e2)
220     | Brela (e1, _, e2) -> (is_boolean func e1) && (is_boolean func e2)
221     | Asn(id, expr2) ->
222         begin
223             let t1 = get_var_type func id and t2 = get_expr_type expr2 func in
224             match t1, t2 with
225             | StringType, StringType -> true
226             | IntType, IntType -> true
227             | DoubleType, DoubleType -> true
228             | ElementType, ElementType -> true (*allow int to double conversion*)
229             | MoleculeType, MoleculeType -> true
230             | EquationType, EquationType -> true
231             | _, _ -> raise(Failure ("DataTypes do not match up in an assignment
232                                     expression to variable "))
233         end
234     | _ -> raise( Failure("!!! Need to implement in valid_expr: Equation,
235                               Concat, Seq, List, Call, Null, Noexpr !!!") )
236
237 (*Print(e1) ->
238     let t1 = get_expr_type expr func in
239     match t1 with
240     | "String" -> true
241     | "int" -> true
242     | "double" -> true
243     | "boolean" -> true
244     | "element" -> true
245     | "molecule" -> true
246     | "equation" -> true
247     | _ -> raise(Failure("Can't print type"))*)
248
249 let has_return_stmt list =
250     if List.length list = 0
251     then false
252     else match (List.hd (List.rev list)) with
253     | Return(_) -> true
254     | _ -> false
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
347     environment *) in
348   (not duplicate_functions) && (* (not duplicate_parameters) && *)
349   v_body
350
351 let check_program flist =
352   let (environment : env) = { functions = [] (* ; variables = [] *) } in
353   let _validate = List.map ( fun f -> valid_func environment f ) flist in
354   (* let _ = print_endline "\nSemantic analysis completed successfully.\n
355     nCompiling...\n" in *)
356   true

```

Listing A.5: Compiler, Code Generation (compile.ml)

```

1  open Ast
2  open Str
3  open Printf
4  open Parser
5  module StringMap = Map.Make(String);;
6
7  let string_of_type = function
8    | IntType -> "int"
9    | BooleanType -> "Boolean"
10   | StringType -> "String"
11   | DoubleType -> "double"
12   | _ -> ""
13
14 let string_of_op = function
15   Add -> "+"
16   | Sub -> "-"
17   | Mul -> "*"
18   | Div -> "/"
19   | Mod -> "%"
20   | Gt -> ">"
21   | Geq -> ">="
22   | Lt -> "<"
23   | Leq -> "<="
24   | Eq -> "=="
25   | Neq -> "!="
26
27 let string_of_re = function
28   And -> "&&"

```

```

29 | Or -> "||"
30
31 let string_of_boolean = function
32   True -> string_of_bool true
33   | False -> string_of_bool false
34
35 let string_of_var = function
36   Var(v)-> v
37
38
39 let string_of_rule = function
40   Balance(equation) -> "Balance(" ^ equation ^ ");"
41   | Mass(equation)-> "Mass(" ^ equation ^ ");"
42
43 let rec string_of_expr = function
44   Int(i) -> string_of_int i
45   | Double(d) -> string_of_float d
46   | Boolean(b) -> string_of_boolean b
47   | String (s) -> s
48   | Asn(id, left) -> id ^ " = " ^ (string_of_expr left)
49   | Seq(s1, s2) -> (string_of_expr s1) ^ " ; " ^ (string_of_expr s2)
50   | Call(s,l) -> s ^ "(" ^ String.concat "" (List.map string_of_expr l) ^ ")"
51   | Access(o,m) -> (string_of_expr o) ^ "." ^ m ^ "();";
52   | 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 ^ " , " ^
53   (string_of_int e1) ^ " , " ^
54   (string_of_int e2) ^ " , " ^
55   (string_of_int e3) ^ " , " ^
56   (string_of_int e4) ^ " , " ^
57   (string_of_int e5) ^ " , " ^
58   (string_of_int e6) ^ " , " ^
59   (string_of_int e7) ^ " , " ^
60   (string_of_int e8) ^ " ) )";
61   | Binop (e1, op, e2) ->
62   (string_of_expr e1) ^ " " ^ (match op with
63     Add -> "+"
64     | Sub -> "-"
65     | Mul -> "*"
66     | Div -> "/"
67     | Mod -> "%"
68     | Gt -> ">"
69     | Geq -> ">="
70     | Lt -> "<"
71     | Leq -> "<="
72     | Eq -> "=="
73     | Neq -> "!=")
74   ^ " " ^ (string_of_expr e2)
75   | Brela (e1, op, e2) ->
76   (string_of_expr e1) ^ " " ^ (match op with

```

```

77     And -> "&&"
78   | Or -> "||"
79   ^ " " ^ (string_of_expr e2)
80   | Noexpr -> ""
81   | Null -> "NULL"
82   | Concat(s1, s2) -> string_of_expr s1 ^ "+" ^ string_of_expr s2
83   | List(elist) -> "[" ^ String.concat "," (List.map string_of_expr elist
84     ) ^ "]"
84   | Print(s) -> "System.out.println(" ^ string_of_expr s ^ ");"
85   | Equation(name, rlist, plist) -> "equation" ^ name ^ "{" ^ String.
86     concat "," (List.map string_of_var rlist) ^ "—" ^ String.concat ","
87     (List.map string_of_var plist) ^ "}"
86   | Bracket(e) -> "(" ^ string_of_expr e ^ ")"
87   (* | Element(name, mass, electron, charge) -> "element " ^ name ^ "(" ^ (
88     string_of_int mass) ^ "," ^ (string_of_int electron) ^ "," ^ (
89     string_of_int charge) ^ ")"
88   | Molecule(name, elist) -> "molecule " ^ name ^ "{" ^ String.concat "," (
89     List.map string_of_var elist) ^ "}" *)
89   let string_of_eddecl eddecl = "Element " ^ eddecl.name ^ "= new Element(" ^ (
90     string_of_int eddecl.mass) ^ "," ^ (string_of_int eddecl.electrons) ^ "," ^
91     (string_of_int eddecl.charge) ^ ");"
90   let string_of_mdecl mdecl = "ArrayList<Element> " ^ mdecl.mname ^ "1 = new
91     ArrayList<Element>(Arrays.asList(" ^ String.concat "," (List.map
92     string_of_var mdecl.elements) ^ "));"
91   "Molecule " ^ mdecl.mname ^ "= new Molecule(" ^ mdecl.mname ^ "1);"
92
93   let string_of_pdecl pdecl = string_of_type pdecl.paramtype ^ " " ^ pdecl.
94     paramname
94   let string_of_pdecl_list pdecl_list = String.concat "" (List.map
95     string_of_pdecl pdecl_list)
95   let string_of_vdecl vdecl = string_of_type vdecl.vtype ^ " " ^ vdecl.vname ^
96     ";\n"
96
97   let rec string_of_stmt = function
98     Block(stmts) ->
99       "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
100   | Expr(expr) -> string_of_expr expr ^ ";\n"
101   | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n"
102   | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n{" ^ (
103     string_of_stmt s) ^ "}"
103   | If(e, s1, s2) -> "if (" ^ string_of_expr e ^ ")\n{" ^ (string_of_stmt
104     s1) ^ "}" ^ "else\n{" ^ (string_of_stmt s2) ^ "}"
104   | For(e1, e2, e3, s) ->
105     "for (" ^ string_of_expr e1 ^ " ; " ^ string_of_expr e2 ^ " ; " ^
106     string_of_expr e3 ^ ") " ^ string_of_stmt s
106   | While(e, s) -> "while (" ^ string_of_expr e ^ ") {" ^ (string_of_stmt s
107     ) ^ "}"
107   | Print(s) -> "System.out.println(" ^ string_of_expr s ^ ");"
108
109
110

```

```

111
112 let string_of_vdecl vdecl =
113     string_of_type vdecl.vtype ^ " " ^ vdecl.vname ^ ";";
114
115 let string_of_fdecl fdecl =
116     if fdecl.fname = "main" then "public static void main(String args[])\n{\n
117         String.concat "" (List.map string_of_vdecl fdecl.locals) ^
118         String.concat "" (List.map string_of_eddecl fdecl.elements) ^
119         String.concat "" (List.map string_of_mdecl fdecl.molecules) ^
120         String.concat "" (List.map string_of_rule fdecl.rules) ^
121         String.concat "" (List.map string_of_stmt fdecl.body) ^
122         "}\n"
123     else
124         "public static void " ^ fdecl.fname ^ "(" ^ String.concat ", " (List.map
125             string_of_pdecl fdecl.formals) ^ ")\n{\n" ^
126         String.concat "" (List.map string_of_vdecl fdecl.locals) ^
127         String.concat "" (List.map string_of_eddecl fdecl.elements) ^
128         String.concat "" (List.map string_of_mdecl fdecl.molecules) ^
129         String.concat "" (List.map string_of_rule fdecl.rules) ^
130         String.concat "" (List.map string_of_stmt fdecl.body) ^
131         "}\n"
132
133 let string_of_fdecl_list fdecl_list =
134     String.concat "" (List.map string_of_fdecl fdecl_list)
135
136 let string_of_program (vars, funcs) =
137     String.concat "" (List.map string_of_vdecl (List.rev vars)) ^ "\n" ^
138     String.concat "\n" (List.map string_of_fdecl (List.rev funcs)) ^ "\n"
139
140 let rec mass_sum element_list = match element_list with
141 | [] -> 0
142 | hd :: tl -> hd.mass + mass_sum tl;;
143
144 let rec charge_sum molecule = match molecule with
145 | [] -> 0
146 | hd :: tl -> hd.charge + charge_sum tl;;
147
148
149 let contains s1 s2 =
150     let re = Str.regexp_string s2
151     in
152         try ignore (Str.search_forward re s1 0); true
153         with Not_found -> false
154
155
156
157
158 let program program prog_name =

```

```

159     let jframe a b =
160     if contains (string_of_fdecl_list program) "graphics" then a else b in
161     let out_chan = open_out ("ChemLAB" ^ ".java") in
162     ignore(Printf.fprintf out_chan
163     "
164     import com.graphics.*;
165     import java.util.*;
166     import java.awt.*;
167     import java.awt.event.*;
168     import java.util.ArrayList;
169     import javax.swing.*;
170
171     public class ChemLAB %s
172     {
173         %s
174         public static boolean debug = false;
175         public static int randx;
176         public static int randy;
177
178         public ChemLAB()
179         {
180             %s
181         }
182
183         public static void Balance(String s)
184         {
185             String [] r = s.split("\\(, )|(==)|(' ')\\");
186             String [] r1 = s.split("\\\\\\s*(,\\\\\\\\s)\\\\\\\\s*\\");
187             String [] r2 = s.split("\\(, )|(' ')\\");
188             String [] individual = s.split("\\(, )|(== )|(?=\\\\\\\\p{Upper})|(' ')\\");
189
190             ArrayList<String> elements = new ArrayList<String>();
191
192             int counter = 0;
193             for (int i=0; i<r2.length; i++){
194                 if (r2[i].contains("\\="\\"))
195                     counter = i;
196             }
197             counter++;
198
199             for (int i = 0; i < individual.length; i++) {
200                 String x = "\\\\";
201                 for (int j = 0; j < individual[i].length(); j++) {
202                     if (Character.isLetter(individual[i].charAt(j)))
203                         x = x + individual[i].charAt(j);
204                 }
205                 if (!elements.contains(x) && (x != "\\\\"))
206                     elements.add(x);
207             }
208

```

```

209 double [][] matrix = new double [elements.size()][r.length];
210
211 for (int i = 0; i < elements.size(); i++) {
212     String temp = elements.get(i);
213     for (int j = 0; j < r.length; j++) {
214         if (r[j].contains(temp)) {
215             int k = r[j].indexOf(temp) + temp.length();
216             if (k >= r[j].length()) {
217                 k = 0;
218             }
219             if (Character.isDigit(r[j].charAt(k))) {
220                 int dig = Integer.parseInt(r[j].substring(k, k + 1));
221                 matrix[i][j] = dig;
222             } else {
223                 matrix[i][j] = 1;
224             }
225         } else {
226             matrix[i][j] = 0;
227         }
228     }
229 }
230
231
232
233 double [][] A = new double[matrix.length][matrix[0].length - 1];
234 double [][] B = new double[matrix.length][1];
235
236 for (int i = 0; i < matrix.length; i++) {
237     for (int j = 0; j < matrix[i].length - 1; j++) {
238         A[i][j] = matrix[i][j];
239     }
240 }
241
242 int n = A[0].length < A.length ? A.length : A[0].length;
243 int difference = Math.abs(A.length - A[0].length);
244 double [][] A1 = new double[n][n];
245
246 for (int i = 0; i < B.length; i++) {
247     B[i][0] = matrix[i][matrix[i].length - 1];
248 }
249
250
251 for (int i = 0; i < A.length; i++)
252 {
253     for (int j = 0; j < A[0].length; j++)
254     {
255         A1[i][j] = A[i][j];
256     }
257 }
258

```



```

259     if (A[0].length < A.length) {
260         for (int i=0; i<n; i++){
261             for (int j = n-difference; j< n; j++)
262                 {
263                     A1[i][j] = 1;
264                 }
265         }
266     }
267     else if (A[0].length > A.length)
268     {
269         for (int i=0; i<n; i++){
270             for (int j = n-difference; j< n; j++)
271                 {
272                     A1[j][i] = 1;
273                 }
274         }
275     }
276
277     for (int i=0; i<n; i++)
278     {
279         for (int j=counter; j<n; j++){
280             matrix[i][j] = matrix[i][j] * -1;
281         }
282     }
283
284     double det = determinant(A1, n);
285     double inverse [][] = invert(A1);
286     double [][] prod = product(inverse, B, det);
287
288     double factor = 0;
289     boolean simplified = true;
290     for (int i = 0; i < prod.length; i++)
291     {
292         for (int j = i; j < prod.length; j++)
293         {
294             if (mod(prod[i][0], prod[j][0]))
295             {
296                 simplified = false;
297                 break;
298             }
299         }
300     }
301
302     if (simplified == false)
303     {
304         factor = findSmallest(prod);
305         simplify(prod, factor);
306     }
307
308     boolean subtract = false;

```

```

309
310     for (int j = 0; j < r1.length; j++)
311     {
312         if (j == r1.length - 1)
313         {
314             int sum = 0;
315             int count = 0;
316             for (int m = 0; m < B[0].length; m++)
317             {
318                 if (B[m][0] == 0)
319                 {
320                     count++;
321                 }
322             }
323             for (int k = 0; k < n; k++)
324             {
325                 sum += Math.round(matrix[count][k] * Math.abs(prod[k][0]));
326             }
327
328             if (B[count][0] == 0)
329             {
330
331                 System.out.println(1 + " " + r2[j - 2]);
332             }
333             else
334             {
335
336                 System.out.println(Math.abs(sum / (int) B[count][0]) + " "
337                     + r2[j - 2]);
338             }
339         }
340         else if (r1[j].equals("\r"))
341         {
342             System.out.print("\r--> ");
343             subtract = true;
344         }
345         else if (subtract == true)
346         {
347             int coeff = (int) Math.round(Math.abs(prod[j - 1][0]));
348             System.out.print(coeff + " " + r1[j] + " ");
349         }
350         else
351         {
352             int coeff = (int) Math.round(Math.abs(prod[j][0]));
353             System.out.print(coeff + " " + r1[j] + " ");
354         }
355     }
356 }
357

```

```

358
359 public static boolean mod(double a, double b)
360 {
361
362     int c = (int)(a)/(int)(b);
363     if (c*b == a)
364         return true;
365     else
366         return false;
367 }
368
369 public static void printMatrix(double [][] matrix)
370 {
371     for (int i = 0; i < matrix.length; i++)
372     {
373         for(int j = 0; j< matrix[0].length; j++)
374         {
375             System.out.print(matrix[i][j] + \" \");
376         }
377         System.out.print(\"\\n\\n\");
378     }
379 }
380
381 public static double findSmallest(double a[][])
382 {
383     double smallest = a[0][0];
384     for(int i = 0; i < a.length; i++)
385     {
386         if(Math.abs(a[i][0]) < Math.abs(smallest))
387             smallest = a[i][0];
388     }
389     return smallest;
390 }
391
392 public static double [][] simplify(double a[][], double smallest)
393 {
394     int largest = 0;
395     boolean all = true;
396     for(int i = 1; i <= Math.abs(smallest); i++)
397     {
398         all = true;
399         for(int j = 0; j < a.length; j++)
400         {
401             if(!mod(a[j][0], i) )
402             {
403                 all = false;
404             }
405         }
406         if (Math.abs(i)>Math.abs(largest) && all == true)
407             largest = i;

```

```

408     }
409     if (debug == true)
410         System.out.println(largest);
411     if(largest!=0)
412     {
413         for(int k = 0; k < a.length; k++)
414         {
415             a[k][0] = a[k][0]/largest;
416         }
417     }
418     return a;
419 }
420
421 public static double [][] product(double a[][], double b[][], double det)
422 {
423     int rowsInA = a.length;
424     int columnsInA = a[0].length; // same as rows in B
425     int columnsInB = b[0].length;
426     double [][] c = new double[rowsInA][columnsInB];
427     for (int i = 0; i < rowsInA; i++) {
428         for (int j = 0; j < columnsInB; j++) {
429             for (int k = 0; k < columnsInA; k++) {
430                 c[i][j] = c[i][j] + a[i][k] * b[k][j];
431             }
432         }
433     }
434
435     for(int i = 0; i < rowsInA; i++)
436     {
437         c[i][0] = c[i][0]*det;
438     }
439     return c;
440 }
441 public static double determinant(double A[][], int N)
442 {
443     double det=0;
444     if(N == 1)
445     {
446         det = A[0][0];
447     }
448     else if (N == 2)
449     {
450         det = A[0][0]*A[1][1] - A[1][0]*A[0][1];
451     }
452     else
453     {
454         det=0;
455         for (int j1=0;j1<N;j1++)
456         {
457             double [][] m = new double[N-1][];

```

```

458         for (int k=0;k<(N-1);k++)
459         {
460             m[k] = new double[N-1];
461         }
462         for (int i=1;i<N;i++)
463         {
464             int j2=0;
465             for (int j=0;j<N;j++)
466             {
467                 if (j == j1)
468                     continue;
469                 m[i-1][j2] = A[i][j];
470                 j2++;
471             }
472         }
473         det += Math.pow(-1.0,1.0+j1+1.0)* A[0][j1] * determinant(m,N-1);
474     }
475 }
476 return det;
477 }
478 public static double [][] invert(double a[][])
479 {
480     int n = a.length;
481     double x[][] = new double[n][n];
482     double b[][] = new double[n][n];
483     int index[] = new int[n];
484     for (int i=0; i<n; ++i)
485         b[i][i] = 1;
486
487     gaussian(a, index);
488
489     for (int i=0; i<n-1; ++i)
490         for (int j=i+1; j<n; ++j)
491             for (int k=0; k<n; ++k)
492                 b[index[j]][k]
493                 -= a[index[j]][i]*b[index[i]][k];
494
495     for (int i=0; i<n; ++i)
496     {
497         x[n-1][i] = b[index[n-1]][i]/a[index[n-1]][n-1];
498         for (int j=n-2; j>=0; --j)
499         {
500             x[j][i] = b[index[j]][i];
501             for (int k=j+1; k<n; ++k)
502             {
503                 x[j][i] -= a[index[j]][k]*x[k][i];
504             }
505             x[j][i] /= a[index[j]][j];
506         }
507     }

```

```

508         return x;
509     }
510
511     // Method to carry out the partial-pivoting Gaussian
512     // elimination. Here index[] stores pivoting order.
513
514     public static void gaussian(double a[][] , int index[])
515     {
516         int n = index.length;
517         double c[] = new double[n];
518
519         // Initialize the index
520         for (int i=0; i<n; ++i)
521             index[i] = i;
522
523         // Find the rescaling factors, one from each row
524         for (int i=0; i<n; ++i)
525         {
526             double c1 = 0;
527             for (int j=0; j<n; ++j)
528             {
529                 double c0 = Math.abs(a[i][j]);
530                 if (c0 > c1) c1 = c0;
531             }
532             c[i] = c1;
533         }
534
535         // Search the pivoting element from each column
536         int k = 0;
537         for (int j=0; j<n-1; ++j)
538         {
539             double pi1 = 0;
540             for (int i=j; i<n; ++i)
541             {
542                 double pi0 = Math.abs(a[index[i]][j]);
543                 pi0 /= c[index[i]];
544                 if (pi0 > pi1)
545                 {
546                     pi1 = pi0;
547                     k = i;
548                 }
549             }
550
551             // Interchange rows according to the pivoting order
552             int itmp = index[j];
553             index[j] = index[k];
554             index[k] = itmp;
555             for (int i=j+1; i<n; ++i)
556             {
557                 double pj = a[index[i]][j]/a[index[j]][j];

```

```

558
559 // Record pivoting ratios below the diagonal
560         a[index[i]][j] = pj;
561
562 // Modify other elements accordingly
563         for (int l=j+1; l<n; ++l)
564             a[index[i]][l] -= pj*a[index[j]][l];
565     }
566 }
567 }
568 %s
569 }" (jframe "extends JFrame" "") (jframe "final static SceneComponent
    scene = new SceneComponent();" "")
570 (jframe "setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); setSize(500,
    500); add(scene, BorderLayout.CENTER);" "") (string_of_fdecl_list
    program ) );
571 close_out out_chan;
572 ignore(Sys.command ("javac ChemLAB.java"));
573 ignore(Sys.command (Printf.sprintf "java %s" "ChemLAB"));
574 let contains s1 s2 =
575 let re = Str.regexp_string s2
576 in
577     try ignore (Str.search_forward re s1 0); true
578     with Not_found -> false
579 in
580     if (contains (string_of_fdecl_list program) "graphics") then
        ignore(Sys.command ("javac ChemLAB.java SceneEditor.java
            ")); ignore(Sys.command("java SceneEditor"));

```

Listing A.6: Top-level Executable (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
28   | NoInputFile -> ignore(Printf.printf "Please provide a name for a
    ChemLAB file.\n");exit 1
29   | InvalidProgram -> ignore(Printf.printf "Invalid program. Semantic
    errors exist.\n");exit 1

```

language=bash]../test.sh