CompA - Complex Analyzer Language

Manager: Xiping Liu(xl2639)
Language guru: Jianshuo Qiu(jq2253)
System Architect: Tianwu Wang(tw2576)
Tester: Yingshuang Zheng(yz3083)
System Architect: Zhanpeng Su(zs2329)

December 20, 2017

Contents

1 Introduction to the Language

1.1 Movivation

Matrices are extraordinarily useful and powerful tools that can be applied to several branches of science and engineering. Running Markov simulations based on stochastic machines, computers can model events from gambling through weather forecasting to quantum mechanics. The use of matrix also greatly simplifies linear algebra by providing more compact ways to solve groups of equations in linear algebra. Matrix also plays an important role in digital sound and digital sound processing. Processing techniques such as filtering or compressing video or audio signals reply on Fourier transform and matrix multiplication. Therefore, our goal is to design a language, CompA, that can both support complex number and matrix operations, and so capable to solve complicated real-life problems such as signal processing and image processing.

1.2 Introduction

Most commonly used programming language allow user to build array like data structures. However, it is always hard to do matrix based operations by using those built-in data structures. For example, you can simulate a matrix by using a 2D array. However, it is inconvenient to do matrix operations by using the built in array functions. Moreover, since complex numbers are also frequently used in matrix related operations and applications, we decide to build our language — CompA that both support complex arithmetic and matrix operations. Our language CompA is the short for Complex Analyser, and it can also be interpreted as possessing A level computation power. CompA has a matrix type and a complex type that allow user to do efficient matrix manipulations and complex arithmetic. For example, users of CompA can create matrix by filling in the values that they want and do operations such as matrix addition, matrix multiplication, transpose and conjugate (determinant) of a matrix. For complex numbers, CompA also supports complex operations such like addition, multiplication, subtraction, exponential, and complex conjugate.

2 Tutorial

2.1 Environment Setup

For environment setup, please refer to the README in the compA folder.

2.2 Generate the compiler

In the compA folder, type 'make' to generate the compa.native file. This file can be used to compile compa code into LLVM code, which can be used in the LLVM compiler to print out a result. Users should follow the syntax and semantics of the language in order to produce a useful program. The instructions will be shown in the sections below.

2.3 First program in CompA

"Hello World" program is always the starting point for each programmer trying a new language. Simple program like "Hello World" in CompA resembles that in C. Create a new file named hello.ca, and use text editor of your choice to write following lines of codes:

```
int main {
    print("Hello World!");
    return 0;
}
```

After that, it is time to compile. The command line for generating output from compa.native compiler is as follows:

```
./compa.native -c hello.ca stdlib.ca | lli
```

note: stdlib.ca is a library of build-in functions for users to access
This command yield an output:

```
Hello World!
```

2.4 The Basics

2.4.1 Datatypes in the Language

The 6 data types are all the built-in data types in our language. Our language is statically-typed. Namely, you must declare the data types of your variables before you use them.

- int
- float
- bool
- string
- complex

We use a 2-tuple surrounded by parentheses to declare complex number, where the first tuple is real part, the second tuple is imaginary part. Both parts must be float.

```
cx a = (1.0,2.0);
```

For matrix, we can declare a 2D 2 by 3 matrix named m1 in the following way.

```
int[2][3] m1;
```

Then, after the declaring the the matrix, we can use our built-in functions written by us in the standard library to populate matrix entries. Notice that the built-in function can only populate one row of the matrix when called.
Hence, a two row 2D matrix needs to call the function twice.
The table below presents more detail of the types in this language.

Type	Declaration
int	int x;
float	float y;
char	char c;
bool	bool b;
complex variable	cx a;
1D int typed Matrix	int[4] m;
2D int typed Matrix	int[2][2] m;
1D Matrix Pointer	int[] p;
2D Matrix Pointer	int[][] p;

Here is an example that we declare and initialize different variables and prints out their values.

```
int main()
{
    float a;
    cx b;
    int c;
    string h;

    a = 3.2;
    b = (3.2,3.4);
    c = 2;
    b[0]= a;
    h = "Hello";

    print(a);
    print(b);
    print(c);
    print(h);

    return 0;
}
```

Output:

```
3.200000
(3.200000,3.400000)
2
Hello
```

There are also some built−in constants in our language.

Constant Name	Mathematical Meaning
PI	Pi approximately 3.14159
INF	infinity
E	Euler's number approximately equals to 2.71828

2.4.2 Control Flow

a. if statement
Handles conditional statements

```
if (<condition>) {
    <statements>
    } else if (<condition>) {
        <statements>
    } else {
            <statements>
}
```

b. for loop
Handles loop operations

```
for (int i = 0; i < 20; i++) {
    <statements>
}
```

c. while loop
Another way to handle loop operations

```
int i = 0;
while (i < 10) {
        <statements>;
        i ++;
}
```

d. break
Terminate a loop(usually with a condition), and the program resumes at the next statement following the loop

```
int i = 0;
while (i < 10) {
        <statements>;
        i++;
    if (i > 6) {
    break;
    }
}
```

e. continue
When a continue statement is encountered inside a loop, control jumps to the beginning of the loop for next iteration, skipping the execution of statements inside the body of loop for the current iteration

```
for (i = 0; i < 10; i++) {
        if (i == 6) {
        continue; // when i = 6, <statements> will be skipped
        and the control will return to the loop with i = 7
        }
        <statements>;
    }
}
```

2.5 Complex Arithmetic Reference

<table>
<tr><th>Complex Number Operators</th><th>Operatio</th><th>Examples</th></tr>
<tr><td>+</td><td>addition</td><td>z1 + z2 = (a + c) + i(b + d)</td></tr>
<tr><td>-</td><td>substraction</td><td>z1 - z2 = (a - c) + i(b - d)</td></tr>
<tr><td>*</td><td>multiplication</td><td>z1 * z2 = (ac - bd) + i(ad + bc)</td></tr>
<tr><td>/</td><td>division</td><td>z1 / z2 =((ac - bd) + i(bc - ad)) / (c^2 + d^2)</td></tr>
<tr><td>^</td><td>power</td><td>z^n = (a + ib)^n</td></tr>
<tr><td>exp()</td><td>exp power</td><td>exp(z) = e^a(cos(b) + i(sin(b)))</td></tr>
<tr><td>conj()</td><td>conjugate</td><td>conj(z)= a - ib</td></tr>
<tr><td>| |</td><td>absolute value</td><td>|z| = (a^2 + b^2)^(1/2)</td></tr>
<tr><td>e</td><td>scientific notation</td><td>5.12e-31 =5.12*10^(-31)</td></tr>
</table>

2.6 Operators & Precedence

Precedence	Operators
1	=
2	——
3	&&
4	==, !=
5	>, <, >=, <=
6	+, -
7	*,/,%
8	!, -

Here a larger number means a higher precedence.

2.7 Comments

Comments are similar to C language, in which /* starts comments and */ ends comments. Anything between /* and */ will be ignored by the syntax. Comments cannot be nested.

2.8 User Defined Function

Users can create their own functions by using primitive data types and built-in functions. The syntax is C-like.
Example1

```
int get_zero(){
    return 0;
}
```

Example2

```
bool age_compare(int age1, int age2){
        if (age1 >= age2){
        return true;
    } else {
            return false;
    }
}
```

Example3

```
mx add_matrix(mx matrix1, mx matrix2){
        return matrix1 + matrix2;
}
```

Example4

```
void printTrace(mx m){
        int trace = tr(m);
        print(trace);
}
```

3 A Sample Program

Below is an example program in our language CompA, which solves a problem in Quantum Mechanics. It uses a user defined function spinXExpectation(int t) to calculate the expectation value of the spin angular momentum in x direction of a wave function at time t. In the main() function, spinXExpectation(int t) is calculated at t = 0 and 5 and the results are printed out to the console window using the built-in function print().

```
/* start of the program */
/* global variables */
static float h_bar = 1.05457e-34;
static float B_0 = 1e-5;
static float alpha = PI/6;
static float gamma = -1.6e-19/9.11e-31

/* main functoin */
int main(mx arg) {
    print("Spin angular momentum in x direction at time t = 0");
    mx expectationValue = spinXExpectation(0);
    print(expectationValue);

    print("Spin angular momentum in x direction at time t = 5");
    expectationValue = spinXExpectation(5);
    print(expectationValue);

    return 0;
}

 /* user defined function */
mx spinXExpectation(float t) {
    mx waveFunction = [cos(alpha/2)exp((0,1)*gamma*B_0*t/2);
    sin(alpha/2)exp(-(0,1)*gamma*B_0*t/2)]; /* complex
    matrix declaration */

    mx S_x = [0, 1; 1, 0];
    return tp(conj(waveFunction))*S_x*waveFunction;/* complex
    matrix multiplication */
}


/* end of the program */
```

4. Project Plan

4.1 Planning Process

Basically, we planned our language during September and October, and started to program our language since late October. We started to do a lot of work since Thanksgiving break, and we made a lot of project since then.

4.2 Style Guide

We used the following conventions while programming our CompA compiler, in order to ensure consistency, readability, and transparency.

- OCaml editing and formatting style to write code for compiler architecture
- C language editing and formatting style for inspiration for CompA program code

A few other style guidelines to note:

- File names end in .ca
- Variable identifiers begin with a lowercase letter and contains letters, numbers, and underscore
- Function identifiers begin with a lowercase letter and contains letters, numbers, and underscore
- Always include a main function in CompA programs

4.3 Time Line

Oct 16: Finished planning and language specification

Nov 8: Successfully complied Hello World

Nov 25: Created test scripts for our features

Dec 6: Complex number feature successfully implemented

Dec 11: Matrix feature successfully implemented

Dec 17: Added standard library functions for complex numbers and matrix

Dec 20: Final Report

4.4 Roles and Responsibilities

Actually, every member of our team participated in all parts of our project. We divided into 2 subgroups, one implementing complex number feature and the other implementing matrix feature. Xiping, Jianshuo, and Tianwu implemented the matrix feature, while Yingshuang and Zhanpeng implemented the complex number feature.

4.5 Software Development Environment

Operating Systems: Mac OS Systems

Languages: OCaml, LLVM

Text Editor: Sublime, Vim

Version Control: GitHub

Documentation: LaTeX, Microsoft Word

5. Architecture Design

The Basic work flow follows:

Scanner--- Parser --- Abstract Syntax Tree --- Code Generation ---LLVM IR --- Executable

5.1 Scanner

The scanner scans CompA source code and parse them into tokens that it recognizes. Comment is ignored during the parsing process while ID literals constant and other keywords are labeled and feed into the compiler waiting for further process these.

5.2 Parser and Abstract Syntax Tree

Parser takes the input coming from scanner and further parse out the important information that compiler need to be used in abstract syntax tree. Abstract Syntax Tree is used to build the structure of the program.

5.3 Semantic Check

Semantic check is important since it helps the programmer to debug easily when writing in CompA. Apart from the primitive types that mentioned before, CompA requires type checking on complex number element type and build in function type. Complex number is built upon float tuples and most of the build in functions are operated on float type. Semantic Check will check throws error if there is a mismatch on any function or variable declaration mismatch and calling type mismatch.

5.4 Code Generation

Code generation is the hardest part during the construction of CompA. The documentation is relatively poorly written. We need to compile equivalent C code to assembly code and compare them to llvm code to figure out the logic behind each function that we are going to implement. Besides, we used llvm intrinsic to link our build in functions for float operations.

MicroC is a good starting point to be built upon code generation. We borrow the basic structure of it and added in our own feature based on the language feature that we want to implement.

5.5 CompA Standard Library

Standard Library of CompA includes basic operations on matrix and complex number which includes basic operations like substraction, addition, multiplication and specific features like conjugate and exponential.

6. Test Plan

Test program 1:

```
int main()
{

  float a;
  float b ;
  b = 3.2;
  a = exp(b);
  println(a);
  return 0;
}
```

Correct Output1:

24.532530

Test program 2:

```
int main()
  {
     float a1;
     cx s2;

     float a2;
     float a3;
     float a4;
     float a5;

     cx c1;
     cx c2;
     cx c3;
     cx c4;
     cx c5;
     cx c6;

     int i1;
     int i2;
     int i3;
     int i4;
     int i5;
```

```
   print("VBbigidiot");

   a1= 93.2;
   a1= sqrt(93.2);
   println(a1);

   a2 = sin(32.2);
   println(a2);

   a3 = cos(98.32);
   println(a3);

   a4 = exp(2.3);
   println(a4);

   a5 = pow(a4,a3);
   println(a5);

   a5 = powi(a4,2);
   println(a5);

   a1 = exp(2.4);
   println(a1);

   a1 = log(2.2);
   println(a1);

    a1 = log10(2.2);
    println(a1);

    a1 = fabs(3.9);
    println(a1);

    a2 = min(2.0,3.0);
    println(a2);

   a2 = max(2.0,3.0);
    println(a2);

   a3 = rnd(a5);
   println(a3);


   return 0;
}
```

Correct Output2;

```
VBbigidiot
9.654015
0.706169
-0.597331
9.974182
0.253128
99.484316
11.023176
0.788457
0.342423
3.900000
2.000000
3.000000
99.000000
```

Test program 3;

```
int main(){
        float[2][1] vector;
        float[2][2] sheer;
        float[2][2] reflect;
        float[2][2] rotate;
        float[2][2] result;
        float[2][2] result2;
        float[][] p;
        int i;
        float a;

        initialize_2D_f(%%vector, 1.0, 2, 1);

        print_2D_f(%%vector, height(vector), width(vector));

        sheer[0][0]= 1.0;
        sheer[0][1]= 1.25;
        sheer[1][0]= 0.2;
        sheer[1][1]= 1.0;

  copy_2D_f(%%sheer, 2, 2, %%reflect, 2, 2);


  println("test copy_2D_f");
        print_2D_f(%%reflect, height(reflect), width(reflect));
```

```
        rotate[0][0]= -1.0;
        rotate[0][1]= 0.0;
        rotate[1][0]= 0.0;
        rotate[1][1]= 1.0;



multiply_2D_f(%%sheer, %%vector, %%result, 2, 2, 2, 1);

println("test multiply_2D_f");
        print_2D_f(%%result, 2, 1);


initialize_2D_f(%%result2, 1.0, 2, 2);
        add_2D_f(%%result2, %%result2, 2, 2);

        println("test add_2D_f");
print_2D_f(%%result2, 2, 2);


        subtract_2D_f(%%result2, %%result2, 2, 2);

        println("test subtract_2D_f");
print_2D_f(%%result2, 2, 2);


/*initialize_2D_f(%%result2, 1.0, 2, 2);*/
add_2D_scalar_f(%%result2, 2.0, 2, 2);

println("test add_2D_scalar_f");
print_2D_f(%%result2, 2, 2);


initialize_2D_f(%%result2, 1.0, 2, 2);
multiply_2D_scalar_f(%%result2, 2.0, 2, 2);

println("test multiply_2D_scalar_f");
print_2D_f(%%result2, 2, 2);



divide_2D_scalar_f(%%result2, 2.0, 2, 2);

println("test divide_2D_scalar_f");
print_2D_f(%%result2, 2, 2);
```

```
    print_2D_f(%%sheer, 2, 2);
    tp_f(%%sheer, %%result2, 2, 2);

    println("test tp_f");
    print_2D_f(%%result2, 2, 2);


    a = tr_f(%%result2, 2, 2);

    println("test tr_f");
    println(a);
    println("");


    a = det_f(%%result2, 2, 2);

    println("test det_f");
    println(a);

    println("test inv_f");
    print_2D_f(%%sheer, 2, 2);
    inv_f(%%sheer, %%result2, 2, 2);
    print_2D_f(%%result2, 2, 2);

}
```

Correct Output3:

```
[  1.000000   ]
[  1.000000   ]

test copy_2D_f
[  1.000000   1.250000   ]
[  0.200000   1.000000   ]

test multiply_2D_f
[  2.250000   ]
[  1.200000   ]
```

test add_2D_f
[2.000000 2.000000]
[2.000000 2.000000]

test subtract_2D_f
[0.000000 0.000000]
[0.000000 0.000000]

test add_2D_scalar_f
[2.000000 2.000000]
[2.000000 2.000000]

test multiply_2D_scalar_f
[2.000000 2.000000]
[2.000000 2.000000]

test divide_2D_scalar_f
[1.000000 1.000000]
[1.000000 1.000000]

[1.000000 1.250000]
[0.200000 1.000000]

test tp_f
[1.000000 0.200000]
[1.250000 1.000000]

test tr_f
2.000000

test det_f
0.750000
test inv_f
[1.000000 1.250000]
[0.200000 1.000000]

[1.333333 -1.666667]
[-0.266667 1.333333]

Explain for tests:
Test1: the first tests shows our built-in functions, in particular exp(), which takes an input of a complex number and generate the result using Euler’s Formula. This test is important that it combines complex number with exponential power, sin() and cos().

Test2: the second test tests almost all of other built-in functions. Most of them are from LLVM.

Test3: this test tests our matrix operations from our rich matrix library. The functions tested includes not only basic matrix addition, multiplication, but also matrix transpose and inverse matrix.

Who did what:

Yingshuang Zheng and Zhanpeng Su together write the complex number part of our project, including Ocaml code, complex number library function, and complex number tests.

Xiping Liu, Jianshuo Qiu, and Tianwu Wang together wrote the matrix part of our project, including Ocaml code, matrix library function, and matrix tests.

All of use wrote the final report and prepared PPT.

8 Appendices

8.1 scanner.mll

```
(* Ocamllex scanner for CompA *)

{ open Parser }

let digit = ['0'-'9']
let ascii = [' '-'!' '#'-'[' ']'-'~']
let string_literal = '"' ((ascii)* as s) '"'
let float = (digit+)['.'](digit+)

rule token = parse
(* Whitespace *)
  [' ' '\t' '\r' '\n'] { token lexbuf }

(* Comments *)
| "/*"      { comment lexbuf }

(* Delimeters *)
| '('       { LPAREN }
| ')'       { RPAREN }
| '{'       { LBRACE }
| '}'       { RBRACE }
| '['       { LSQRBR }
| ']'       { RSQRBR }
| ';'       { SEMI }
| ','       { COMMA }

(* Operators *)
| '+'       { PLUS }
| '-'       { MINUS }
| '*'       { TIMES }
| '/'       { DIVIDE }
| '='       { ASSIGN }

(* Logical Operators *)
| "=="      { EQ }
| "!="      { NEQ }
| '<'       { LT }
| "<="      { LEQ }
| ">"       { GT }
| ">="      { GEQ }
| "&&"      { AND }
| "||"      { OR }
```

```
| "!"      { NOT }

(* Reference Dereference *)
| '%'      { PERCENT }
| '#'      { OCTOTHORP }

(* Matrices *)
| "len"    { LEN }
| "row"    { ROW }
| "col"    { COL }

(* Control Flow *)
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }

(* Data Types *)
| "int"    { INT }
| "bool"   { BOOL }
| "string" { STRING }
| "float"  { FLOAT }
| "cx"     { COMPLEX }
| "void"   { VOID }

(* Data Values *)
| "true"   { TRUE }
| "false"  { FALSE }
| "PI"     { PI }
| string_literal { STRLIT(s) }
| float as lxm { FLOATLIT(float_of_string lxm) }
| digit+ as lxm { INTLIT(int_of_string lxm) }
| digit*'.'digit+ as lxm { FLOATLIT(float_of_string lxm) }

(* Identifiers *)
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }

(* End of File and Invalid Characters *)
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

8.2 parser.mll

```
%{
open Ast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA LSQRBR RSQRBR
%token PLUS MINUS TIMES DIVIDE ASSIGN NOT
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE PI AND OR
%token RETURN IF ELSE FOR WHILE INT FLOAT BOOL STRING VOID COMPLEX
%token <int> INTLIT
%token <float> FLOATLIT
%token <string> ID STRLIT
%token EOF
%token LEN ROW COL PERCENT OCTOTHORP

%nonassoc NOELSE
%nonassoc ELSE
%nonassoc NOLSQRBR
%nonassoc LSQRBR
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%right NOT NEG

%start program
%type <Ast.program> program

%%

program:
  decls EOF { $1 }

decls:
   /* nothing */ { [], [] }
 | decls vdecl { ($2 :: fst $1), snd $1 }
 | decls fdecl { fst $1, ($2 :: snd $1) }

fdecl:
   typ ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
     { { typ = $1;
```

```
   fname = $2;
   formals = $4;
   locals = List.rev $7;
   body = List.rev $8 } }

formals_opt:
    /* nothing */ { [] }
  | formal_list   { List.rev $1 }

formal_list:
    typ ID                    { [($1,$2)] }
  | formal_list COMMA typ ID { ($3,$4) :: $1 }

typ:
    INT { Int }
  | FLOAT { Float }
  | STRING { String }
  | BOOL { Bool }
  | VOID { Void }
  | COMPLEX { Complex }
  | matrix1D_typ { $1 }
  | matrix2D_typ { $1 }
  | matrix1D_pointer_typ { $1 }
  | matrix2D_pointer_typ { $1 }

matrix1D_typ:
    typ LSQRBR INTLIT RSQRBR %prec NOLSQRBR  { Matrix1DType($1, $3) }

matrix2D_typ:
    typ LSQRBR INTLIT RSQRBR LSQRBR INTLIT RSQRBR  { Matrix2DType($1,
   $3, $6) }

matrix1D_pointer_typ:
  typ LSQRBR RSQRBR %prec NOLSQRBR { Matrix1DPointer($1)}

matrix2D_pointer_typ:
  typ LSQRBR RSQRBR LSQRBR RSQRBR { Matrix2DPointer($1) }

vdecl_list:
    /* nothing */    { [] }
  | vdecl_list vdecl { $2 :: $1 }

vdecl:
   typ ID SEMI { ($1, $2) }

stmt_list:
```

```
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
    expr SEMI { Expr $1 }
  | RETURN SEMI { Return Noexpr }
  | RETURN expr SEMI { Return $2 }
  | LBRACE stmt_list RBRACE { 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_opt SEMI expr SEMI expr_opt RPAREN stmt
     { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }

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

expr:
    primitives       { $1 }
  | STRLIT           { StrLit($1) }
  | TRUE             { BoolLit(true) }
  | FALSE            { BoolLit(false) }
  | ID               { Id($1) }
  | PI               { FloatLit(3.14159265358979323846264338327950)}
  | ID LSQRBR expr RSQRBR { ComplexAccess($1, $3) }
  | ID LSQRBR expr RSQRBR ASSIGN expr { Cxassign($1, $3, $6) }
  | LPAREN expr COMMA expr RPAREN { Cx($2,$4) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater, $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
  | expr AND    expr { Binop($1, And,   $3) }
  | expr OR     expr { Binop($1, Or,    $3) }
  | MINUS expr %prec NEG { Unop(Neg, $2) }
  | NOT expr         { Unop(Not, $2) }
  | expr ASSIGN expr   { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { $2 }
  | LSQRBR matrix_literal RSQRBR                  { MatrixLiteral(List.
```

```
    rev $2) }
  | ID LSQRBR expr  RSQRBR %prec NOLSQRBR          { Matrix1DAccess($1,
    $3)}
  | ID LSQRBR expr  RSQRBR LSQRBR expr  RSQRBR     { Matrix2DAccess($1,
    $3, $6)}
  | PERCENT ID                                     { Matrix1DReference(
    $2)}
  | PERCENT PERCENT ID                             { Matrix2DReference(
    $3)}
  | OCTOTHORP ID                                   { Dereference($2)}
  | PLUS PLUS ID                                   { PointerIncrement($3
    ) }
  | LEN LPAREN ID RPAREN                           { Len($3) }
  | ROW LPAREN ID RPAREN                           { Row($3) }
  | COL LPAREN ID RPAREN                           { Col($3) }

primitives:
    INTLIT           { IntLit($1) }
  | FLOATLIT         { FloatLit($1) }

matrix_literal:
    primitives                       { [$1] }
  | matrix_literal COMMA primitives { $3 :: $1 }

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

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

8.3 ast.ml

```
(* Abstract Syntax Tree and functions for printing it *)

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

type uop = Neg | Not

type typ = Int | Bool | Void | String | Float | Complex | Illegal
    | Matrix1DType of typ * int
```

```
    | Matrix2DType of typ * int * int
    | Matrix1DPointer of typ
    | Matrix2DPointer of typ

type bind = typ * string

type expr =
    IntLit of int
  | FloatLit of float
  | StrLit of string
  | BoolLit of bool
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of expr * expr
  | Call of string * expr list
  | Cx of expr * expr
  | ComplexAccess of string * expr
  | Cxassign of string * expr * expr
  | Noexpr
  | PointerIncrement of string
  | MatrixLiteral of expr list
  | Matrix1DAccess of string * expr
  | Matrix2DAccess of string * expr * expr
  | Matrix1DReference of string
  | Matrix2DReference of string
  | Dereference of string
  | Len of string
  | Row of string
  | Col of string

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt

type func_decl = {
    typ : typ;
    fname : string;
    formals : bind list;
    locals : bind list;
    body : stmt list;
  }
```

```
type program = bind list * func_decl list

(* Pretty-printing functions *)

let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"


let string_of_uop = function
    Neg -> "-"
  | Not -> "!"


let string_of_matrix m =
  let rec string_of_matrix_lit = function
      [] -> "]"
    | [hd] -> (match hd with
                IntLit(i) -> string_of_int i
              | FloatLit(i) -> string_of_float i
              | BoolLit(i) -> string_of_bool i
              | Id(s) -> s
              | _ -> raise( Failure("Illegal expression in matrix
    literal") )) ^ string_of_matrix_lit []
    | hd::tl -> (match hd with
                    IntLit(i) -> string_of_int i ^ ", "
                  | FloatLit(i) -> string_of_float i ^ ", "
                  | BoolLit(i) -> string_of_bool i ^ ", "
                  | Id(s) -> s
                  | _ -> raise( Failure("Illegal expression in matrix
    literal") )) ^ string_of_matrix_lit tl
  in
  "[" ^ string_of_matrix_lit m


```

```
let rec string_of_expr = function
    IntLit(l) -> string_of_int l
  | FloatLit(f) -> string_of_float f
  | StrLit(s) -> s
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | FloatLit(3.14159265358979323846264338327950) -> string_of_float 3.14159265358979323846264338327950
  | Id(s) -> s
  | Cx(e1, e2) -> "(" ^ string_of_expr e1 ^ "," ^ string_of_expr e2 ^ ")"
  | ComplexAccess(s, e1) -> s ^ "[" ^ string_of_expr e1 ^ "]"
  | Cxassign(v, e1, e2) -> v ^ "[" ^ string_of_expr e1 ^ "] = " ^ string_of_expr e2
  | Binop(e1, o, e2) -> string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
  | Assign(e1, e2) -> (string_of_expr e1) ^ " = " ^ (string_of_expr e2)
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""

  | PointerIncrement(s) -> "++" ^ s
  | MatrixLiteral(m) -> string_of_matrix m
  | Matrix1DAccess(s, r1) -> s ^ "[" ^ (string_of_expr r1) ^ "]"
  | Matrix2DAccess(s, r1, r2) -> s ^ "[" ^ (string_of_expr r1) ^ "]" ^ "[" ^ (string_of_expr r2) ^ "]"
  | Matrix1DReference(s) -> "%" ^ s
  | Matrix2DReference(s) -> "%%" ^ s
  | Dereference(s) -> "#" ^ s
  | Len(s) -> "len(" ^ s ^ ")"
  | Row(s) -> "row(" ^ s ^ ")"
  | Col(s) -> "col(" ^ s ^ ")"

let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) -> "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
      string_of_expr e3  ^ ") " ^ string_of_stmt s
```

```
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s

let rec string_of_typ = function
    Int -> "int"
  | Float -> "float"
  | Bool -> "bool"
  | Void -> "void"
  | String -> "string"
  | Complex -> "cx"
  | Matrix1DType(t, i1) -> string_of_typ t ^ "[" ^ string_of_int i1 ^ "]"
  | Matrix2DType(t, i1, i2) -> string_of_typ t ^ "[" ^ string_of_int i1 ^ "]" ^ "[" ^ string_of_int i2 ^ "]"
  | Matrix1DPointer(t) -> string_of_typ t ^ "[]"
  | Matrix2DPointer(t) -> string_of_typ t ^ "[][]"

let string_of_vdecl (t, id) = string_of_typ t ^ " " ^ id ^ ";\n"

let string_of_fdecl fdecl =
  string_of_typ fdecl.typ ^ " " ^
  fdecl.fname ^ "(" ^ String.concat ", " (List.map snd fdecl.formals) ^
  ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_program (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

8.4 codegen.ml

```
(* Code generation: translate takes a semantically checked AST and
produces LLVM IR

LLVM tutorial: Make sure to read the OCaml version of the tutorial

http://llvm.org/docs/tutorial/index.html

Detailed documentation on the OCaml LLVM library:

http://llvm.moe/
http://llvm.moe/ocaml/
```

```
*)

module L = Llvm
module A = Ast
module Semant = Semant
open Exceptions


module StringMap = Map.Make(String)

let translate (globals, functions) =
  let context = L.global_context () in
  let the_module = L.create_module context "CompA"
  and i32_t  = L.i32_type  context
  and i8_t   = L.i8_type   context
  and i1_t   = L.i1_type   context
  and str_t  = L.pointer_type (L.i8_type context)
  and pointer_t = L.pointer_type
  and array_t   = L.array_type
  and void_t = L.void_type context
  and f32_t   = L.float_type context in
  let f32x4_t = L.vector_type f32_t 4
  and float_t = L.double_type context in
  let cx_t = L.array_type float_t 2 in

  (*let cx_pointer_t = L.pointer_type float_t in*)
  (*let cx_fst = L.extractvalue cx_t 1 in*)
  (*let cx_snd = L.extractvalue cx_t 2 in*)

let ltype_of_typ = function
      A.Int -> i32_t
    | A.Float -> float_t
    | A.String -> str_t
    | A.Bool -> i1_t
    | A.Void -> void_t
    | A.Matrix1DType(typ, size) -> (match typ with
                                      A.Int -> array_t i32_t size
                                    | A.Float -> array_t float_t size
                                    | A.Bool -> array_t i1_t size
                                    | A.Matrix2DType(typ, size1, size2) -> (match typ with
              A.Int -> array_t (array_t i32_t size2) size1
```

```
             | A.Float -> array_t (array_t float_t size2) size1

             | _ -> raise ( UnsupportedMatrixType )

           )
                                            | _ -> raise (
    UnsupportedMatrixType )
                                           )
      | A.Matrix2DType(typ, size1, size2) -> (match typ with
                                              A.Int -> array_t (array_t
    i32_t size2) size1
                                            | A.Float -> array_t (array_t
     float_t size2) size1
                                            | A.Matrix1DType(typ1, size3)
     -> (match typ1 with

       | A.Int -> array_t (array_t (array_t i32_t size3) size2) size1

       | A.Float -> array_t (array_t (array_t float_t size3) size2) size1

       | _ -> raise (UnsupportedMatrixType)

     )
                                            | _ -> raise (
    UnsupportedMatrixType )
                                           )
      | A.Matrix1DPointer(t) -> (match t with
                                   A.Int -> pointer_t i32_t
                                 | A.Float -> pointer_t float_t
                                 | _ -> raise (IllegalPointerType))
      | A.Matrix2DPointer(t) -> (match t with
                                   A.Int -> pointer_t i32_t
                                 | A.Float -> pointer_t float_t
                                 | _ -> raise (IllegalPointerType))
      | A.Complex -> cx_t in

  (*
  let pointer_wrapper =
    List.fold_left (fun m name -> StringMap.add name (L.
   named_struct_type context name) m)
    StringMap.empty ["string"; "int"; "void"; "bool"]
  in
  (* Set the struct body (fields) for each of the pointer struct types
   *)
  List.iter2 (fun n l -> let t = StringMap.find n pointer_wrapper in
  ignore(L.struct_set_body t (Array.of_list(l)) true))
```

```
["int"; "string"; "void"; "bool"]
[[L.pointer_type i32_t; i32_t; i32_t];
[L.pointer_type str_t; i32_t; i32_t];
[L.pointer_type void_t; i32_t; i32_t]; [L.pointer_type i1_t; i32_t;
 i32_t]];
*)

(* Declare each global variable; remember its value in a map *)

let global_vars =
  let global_var m (t, n) =
    let init = L.const_int (ltype_of_typ t) 0
    in StringMap.add n (L.define_global n init the_module) m in
  List.fold_left global_var StringMap.empty globals in



(* Declare printf(), which the print built-in function will call *)
let printf_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |
 ] in
let printf_func = L.declare_function "printf" printf_t the_module in

let sqrtps = L.declare_function "llvm.sqrt.f64"
   (L.function_type float_t [|float_t|]) the_module in

let sinps = L.declare_function "llvm.sin.f64"
   (L.function_type float_t [|float_t|]) the_module in

let cosps = L.declare_function "llvm.cos.f64"
   (L.function_type float_t [|float_t|]) the_module in

let powips = L.declare_function "llvm.powi.f64"
   (L.function_type float_t [|float_t; i32_t |]) the_module in

let powps = L.declare_function "llvm.pow.f64"
   (L.function_type float_t [|float_t; float_t |]) the_module in


let expps = L.declare_function "llvm.exp.f64"
   (L.function_type float_t [|float_t |]) the_module in

let logps = L.declare_function "llvm.log.f64"
   (L.function_type float_t [|float_t |]) the_module in

let log10ps = L.declare_function "llvm.log10.f64"
   (L.function_type float_t [|float_t |]) the_module in
```

```
let fabsps = L.declare_function "llvm.fabs.f64"
   (L.function_type float_t [|float_t |]) the_module in

let minps = L.declare_function "llvm.minnum.f64"
   (L.function_type float_t [|float_t;float_t  |]) the_module in

let maxps = L.declare_function "llvm.maxnum.f64"
   (L.function_type float_t [|float_t;float_t|]) the_module in

let roundps = L.declare_function "llvm.trunc.f64"
   (L.function_type float_t [|float_t|]) the_module in

(*let printcx_t = L.var_arg_function_type i32_t [| cx_fst;cx_snd |] in
let printcx_func = L.declare_function "printcx" printf_t the_module in*)

(*let s = build_global_stringptr "Hello, world!\n" "" builder in
let zero = const_int i32_t 0 in
let s = build_in_bounds_gep s [| zero |] "" builder in
let _ = build_call printf [| s |] "" builder in
let _ = build_ret (const_int i32_t 0) builder in
*)


(* Define each function (arguments and return type) so we can call it *)
let function_decls =
  let function_decl m fdecl =
    let name = fdecl.A.fname
    and formal_types =
Array.of_list (List.map (fun (t,_) -> ltype_of_typ t) fdecl.A.formals)
    in let ftype = L.function_type (ltype_of_typ fdecl.A.typ) formal_types in
    StringMap.add name (L.define_function name ftype the_module, fdecl) m in
  List.fold_left function_decl StringMap.empty functions in
```

```
(* Fill in the body of the given function *)
let build_function_body fdecl =
  let (the_function, _) = StringMap.find fdecl.A.fname function_decls in
  let builder = L.builder_at_end context (L.entry_block the_function) in

  let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder in
  let float_format_str = L.build_global_stringptr "%f\n" "fmt" builder in
  let str_format_str = L.build_global_stringptr "%s\n" "fmt" builder in
  let cx_format_str = L.build_global_stringptr "(%f,%f)\n" "fmt" builder in
  let intl_format_str = L.build_global_stringptr "%d" "fmt" builder in
  let floatl_format_str = L.build_global_stringptr "%f" "fmt" builder in
  let strl_format_str = L.build_global_stringptr "%s" "fmt" builder in
  let cxl_format_str = L.build_global_stringptr "(%f,%f)" "fmt" builder in

  (* Construct the function's "locals": formal arguments and locally
     declared variables.  Allocate each on the stack, initialize their
     value, if appropriate, and remember their values in the "locals" map *)
  let local_vars =
    let add_formal m (t, n) p = L.set_value_name n p;
let local = L.build_alloca (ltype_of_typ t) n builder in
ignore (L.build_store p local builder);
StringMap.add n local m in

    let add_local m (t, n) =
let local_var = L.build_alloca (ltype_of_typ t) n builder
in StringMap.add n local_var m in

    let formals = List.fold_left2 add_formal StringMap.empty fdecl.A.formals
        (Array.to_list (L.params the_function)) in
    List.fold_left add_local formals fdecl.A.locals in
```

```

  (* Return the value for a variable or formal argument *)
  let lookup n = try StringMap.find n local_vars
                 with Not_found -> StringMap.find n global_vars in
  let check_func =
        List.fold_left(fun m(t,n) -> StringMap.add n t m )
        StringMap.empty(globals@fdecl.A.formals@fdecl.A.locals)
  in
  let type_of_identifier s =
      let symbols = check_func in StringMap.find s symbols in



  let build_complex_argument s builder =
          L.build_in_bounds_gep (lookup s) [| L.const_int i32_t 0; L.const_int i32_t 0|] s builder
  in

  let build_complex_access s i1 i2 builder  =
          L.build_load (L.build_gep (lookup s) [| i1; i2|] s builder) s builder
  in

  let build_complex_real s builder  =
          L.build_load (L.build_gep (lookup s) [| L.const_int i32_t 0;L.const_int i32_t 0 |] s builder) s builder
  in



  let build_1D_matrix_argument s builder =
    L.build_in_bounds_gep (lookup s) [| L.const_int i32_t 0; L.const_int i32_t 0 |] s builder
  in

  let build_2D_matrix_argument s builder =
    L.build_in_bounds_gep (lookup s) [| L.const_int i32_t 0; L.const_int i32_t 0; L.const_int i32_t 0 |] s builder
  in


  let build_1D_matrix_access s i1 i2 builder isAssign =
    if isAssign
      then L.build_gep (lookup s) [| i1; i2 |] s builder
    else
       L.build_load (L.build_gep (lookup s) [| i1; i2 |] s builder) s
```

```
     builder
    in

    let build_2D_matrix_access s i1 i2 i3 builder isAssign =
      if isAssign
        then L.build_gep (lookup s) [| i1; i2; i3 |] s builder
      else
         L.build_load (L.build_gep (lookup s) [| i1; i2; i3 |] s builder) s builder
    in

    let build_pointer_dereference s builder isAssign =
      if isAssign
        then L.build_load (lookup s) s builder
      else
        L.build_load (L.build_load (lookup s) s builder) s builder
    in

    let build_pointer_increment s builder isAssign =
      if isAssign
        then L.build_load (L.build_in_bounds_gep (lookup s) [| L.const_int i32_t 1 |] s builder) s builder
      else
        L.build_in_bounds_gep (L.build_load (L.build_in_bounds_gep (lookup s) [| L.const_int i32_t 0 |] s builder) s builder) [| L.const_int i32_t 1 |] s builder
    in

    let rec matrix_expression e =
       match e with
       | A.IntLit i -> i
       | A.Binop (e1, op, e2) -> (match op with
              A.Add     -> (matrix_expression e1) + (matrix_expression e2)
            | A.Sub     -> (matrix_expression e1) - (matrix_expression e2)
            | A.Mult    -> (matrix_expression e1) * (matrix_expression e2)
            | A.Div     -> (matrix_expression e1) / (matrix_expression e2)
            | _ -> 0)
       | _ -> 0
    in

    let find_matrix_type matrix =
      match (List.hd matrix) with
```

```
        A.IntLit _ -> ltype_of_typ (A.Int)
      | A.FloatLit _ -> ltype_of_typ (A.Float)
      | _ -> raise (UnsupportedMatrixType) in



let rec check_type = function
 A.IntLit _ -> A.Int
|A.FloatLit _-> A.Float
|A.StrLit _-> A.String
     | A.BoolLit _ -> A.Bool
     | A.Id s -> type_of_identifier s
     | A.Cx(e1,e2) -> let t1 =  check_type e1 and t2 = check_type e2 in ( match t1 with A.Float when t2= A.Float -> A.Complex)
     | A.Binop(e1, op, e2) -> let t1 = check_type e1 and t2 = check_type e2 in
(match op with
        Add | Sub | Mult | Div when t1 = A.Int && t2 = A.Int -> A.Int
| Add | Sub | Mult | Div when t1 = A.Complex && t2 = A.Complex -> A.Complex
| Add | Sub | Mult | Div when t1 = A.Float && t2 = A.Float -> A.Float
| Equal | Neq when t1 = t2 -> A.Bool
| Less | Leq | Greater | Geq when t1 = A.Int && t2 = A.Int -> A.Bool
| And | Or when t1 = A.Bool && t2 = A.Bool -> A.Bool
      | _ -> A.Illegal
)
| A.Unop(op, e)  -> let t = check_type e in
 (match op with
   Neg when t = A.Int -> A.Int
 | Not when t = A.Bool -> A.Bool
 | Neg when t = A.Complex -> A.Complex
 | Neg when t = A.Float -> A.Float
 | _ -> Illegal)
 | A.Noexpr -> A.Void
 | A.Assign(_, e)  -> check_type e
 | A.Call(fname, actuals) as call -> A.Illegal
 | A.ComplexAccess(s, c) -> A.Float

 | PointerIncrement(s) -> A.Float
 | MatrixLiteral s -> A.Float
 | Matrix1DAccess(s, e1) -> A.Float
 | Matrix2DAccess(s, e1, e2) -> A.Float
 | Len(s) -> A.Int
 | Row(s) -> A.Int
 | Col(s) -> A.Int
 | Dereference(s) -> A.Float
```

```
  | Matrix1DReference(s) -> A.Float
  | Matrix2DReference(s) -> A.Float

 in


   (* Construct code for an expression; return its value *)
   let rec expr builder = function
 (*TODO*)
   (* A.Literal i -> L.const_int i32_t i*)
   A.IntLit i -> L.const_int i32_t i
     | A.FloatLit f -> L.const_float float_t f
     | A.StrLit s -> L.build_global_stringptr s "string" builder
     | A.BoolLit b -> L.const_int i1_t (if b then 1 else 0)
     | A.Noexpr -> L.const_int i32_t 0
     | A.Id s -> L.build_load (lookup s) s builder
     | A.Cx(e1,e2) ->
     let e1' = expr builder e1
     and e2' = expr builder e2 in L.const_array float_t [|e1';e2'|]
     | A.ComplexAccess (s, e) -> let i1 = expr builder e in (match (type_of_identifier s) with
                                            A.Complex -> (build_complex_access s (L.const_int i32_t 0) i1 builder)
                                            | _ -> build_complex_access s (L.const_int i32_t 0) i1 builder)

     | A.MatrixLiteral s -> L.const_array (find_matrix_type s) (Array.of_list (List.map (expr builder) s))
     | A.Matrix1DReference (s) -> build_1D_matrix_argument s builder
     | A.Matrix2DReference (s) -> build_2D_matrix_argument s builder
     | A.Len s -> (match (type_of_identifier s) with A.Matrix1DType(_, l) -> L.const_int i32_t l
                                                     | _ -> L.const_int i32_t 0 )
     | A.Row s -> (match (type_of_identifier s) with A.Matrix2DType(_, l, _) -> L.const_int i32_t l
                                                     | _ -> L.const_int i32_t 0 )
     | A.Col s -> (match (type_of_identifier s) with A.Matrix2DType(_, _, l) -> L.const_int i32_t l
                                                     | _ -> L.const_int i32_t 0 )
     | A.Matrix1DAccess (s, e1) -> let i1 = expr builder e1 in (match (type_of_identifier s) with
                                                     A.Matrix1DType(_, l) -> (
```

```
                                                      if (matrix_expression e1) >= l then raise(MatrixOutOfBounds)
                                                      else build_1D_matrix_access s (L.const_int i32_t 0) i1 builder false)
                                                    | _ -> build_1D_matrix_access s (L.const_int i32_t 0) i1 builder false )
   | A.Matrix2DAccess (s, e1, e2) -> let i1 = expr builder e1 and i2 = expr builder e2 in (match (type_of_identifier s) with
                                                    A.Matrix2DType(_, l1, l2) -> (
                                                      if (matrix_expression e1) >= l1 then raise(MatrixOutOfBounds)
                                                      else if (matrix_expression e2) >= l2 then raise(MatrixOutOfBounds)
                                                      else build_2D_matrix_access s (L.const_int i32_t 0) i1 i2 builder false)
                                                    | _ -> build_2D_matrix_access s (L.const_int i32_t 0) i1 i2 builder false )
   | A.PointerIncrement (s) ->  build_pointer_increment s builder false
   | A.Dereference (s) -> build_pointer_dereference s builder false
   | A.Binop (e1, op, e2) ->
    let e1' = expr builder e1
    and e2' = expr builder e2
    and typ = check_type e1 in
    (if typ = A.Float then
 (match op with
   A.Add     -> L.build_fadd
 | A.Sub     -> L.build_fsub
 | A.Mult    -> L.build_fmul
 | A.Div     -> L.build_fdiv
 | A.And     -> L.build_and
 | A.Or      -> L.build_or
 | A.Equal   -> L.build_fcmp L.Fcmp.Oeq
 | A.Neq     -> L.build_fcmp L.Fcmp.One
 | A.Less    -> L.build_fcmp L.Fcmp.Ult
 | A.Leq     -> L.build_fcmp L.Fcmp.Ole
 | A.Greater -> L.build_fcmp L.Fcmp.Ogt
 | A.Geq     -> L.build_fcmp L.Fcmp.Oge
 ) e1' e2' "tmp" builder else (match op with
   A.Add     -> L.build_add
 | A.Sub     -> L.build_sub
 | A.Mult    -> L.build_mul
 | A.Div     -> L.build_sdiv
 | A.And     -> L.build_and
 | A.Or      -> L.build_or
```

```
    | A.Equal   -> L.build_icmp L.Icmp.Eq
    | A.Neq     -> L.build_icmp L.Icmp.Ne
    | A.Less    -> L.build_icmp L.Icmp.Slt
    | A.Leq     -> L.build_icmp L.Icmp.Sle
    | A.Greater -> L.build_icmp L.Icmp.Sgt
    | A.Geq     -> L.build_icmp L.Icmp.Sge
    ) e1' e2' "tmp" builder)
| A.Unop(op, e) ->
    let e' = expr builder e in
    let t = check_type e in
    (match op with
        A.Neg when t = A.Int -> L.build_neg
      | A.Neg when t = A.Float -> L.build_fneg
      | A.Not     -> L.build_not) e' "tmp" builder

      | A.Assign (e1, e2) -> let e1' = (match e1 with
                                          A.Id s -> lookup s
                                        | A.Matrix1DAccess (s, e1) -> let i1 = expr builder e1 in (match (type_of_identifier s) with
                                                    A.Matrix1DType(_, l) -> (
                                                      if (matrix_expression e1) >= l then raise(MatrixOutOfBounds)
                                                      else build_1D_matrix_access s (L.const_int i32_t 0) i1 builder true)
                                                    | _ -> build_1D_matrix_access s (L.const_int i32_t 0) i1 builder true )
                                        | A.Matrix2DAccess (s, e1, e2) -> let i1 = expr builder e1 and i2 = expr builder e2 in (match (type_of_identifier s) with
                                                    A.Matrix2DType(_, l1, l2) -> (
                                                      if (matrix_expression e1) >= l1 then raise(MatrixOutOfBounds)
                                                      else if (matrix_expression e2) >= l2 then raise(MatrixOutOfBounds)
                                                      else build_2D_matrix_access s (L.const_int i32_t 0) i1 i2 builder true)
                                                    | _ -> build_2D_matrix_access s (L.const_int i32_t 0) i1 i2 builder true )
                                        | A.PointerIncrement(s) -> build_pointer_increment s builder true
                                        | A.Dereference(s) -> build_pointer_dereference s builder true
                                        | _ -> raise (IllegalAssignment)
```

```
                                          )
                           and e2' = expr builder e2 in
                    ignore (L.build_store e2' e1' builder); e2'
    (*| A.Assign (s, e) -> let e' = expr builder e in
                    ignore (L.build_store e' (lookup s) builder); e'*)

    | A.Cxassign (s,e1,e2) ->  let e1' = expr builder e1
                                and e2' = expr builder e2 in
                                let comp = L.build_gep (lookup s) [| L
.const_int i32_t 0 ; e1'|] s builder in
                                ignore (L.build_store e2' comp builder
); e2'
    | A.Call ("println", [e]) | A.Call ("printb", [e]) -> (match
check_type e with
    A.Float -> L.build_call printf_func [| float_format_str ; (expr
builder e) |]
    "printf" builder
    |A.Int -> L.build_call printf_func [| int_format_str ; (expr
builder e) |]
    "printf" builder
    |A.String -> L.build_call printf_func [| str_format_str ; (expr
builder e) |]
    "printf" builder
    |A.Complex  ->
     L.build_call printf_func [| cx_format_str ; (expr builder e)|]
    "printf" builder
    )
    |A.Call ("print", [e]) ->(match check_type e with
     A.Float -> L.build_call printf_func [| floatl_format_str ; (expr
 builder e) |]
    "printf" builder
    |A.Int -> L.build_call printf_func [| intl_format_str ; (expr
builder e) |]
    "printf" builder
    |A.String -> L.build_call printf_func [| strl_format_str ; (expr
builder e) |]
    "printf" builder
    |A.Complex  ->
     L.build_call printf_func [| cxl_format_str ; (expr builder e)|]
    "printf" builder
    )
    |A.Call ("sqrt", [e1])  -> L.build_call sqrtps [| (expr builder
e1)|] "sqrt" builder
    |A.Call ("sin", [e1])  -> L.build_call sinps [| (expr builder e1)
|] "sin" builder
    |A.Call ("cos", [e1])  -> L.build_call cosps [| (expr builder e1)
```

```
|] "cos" builder
   |A.Call ("powi", [e1;e2])  -> L.build_call powips [| (expr
builder e1);(expr builder e2)|] "powi" builder
   |A.Call ("pow", [e1;e2])  -> L.build_call powps [| (expr builder
e1);(expr builder e2)|] "pow" builder
   |A.Call ("exp", [e1])  -> L.build_call expps [| (expr builder e1)
|] "exp" builder
   |A.Call ("log", [e1])  -> L.build_call logps [| (expr builder e1)
|] "log" builder
   |A.Call ("log10", [e1])  -> L.build_call log10ps [| (expr builder
 e1)|] "log10" builder
   |A.Call ("fabs", [e1])  -> L.build_call fabsps [| (expr builder
e1)|] "fabs" builder
   |A.Call ("min", [e1;e2])  -> L.build_call minps [| (expr builder
e1);(expr builder e2)|] "fabs" builder
   |A.Call ("max", [e1;e2])  -> L.build_call maxps [| (expr builder
e1);(expr builder e2)|] "max" builder
   |A.Call ("rnd", [e1])  -> L.build_call roundps [| (expr builder
e1)|] "rnd" builder
   | A.Call (f, act) ->
     let (fdef, fdecl) = StringMap.find f function_decls in
 let actuals = List.rev (List.map (expr builder) (List.rev act)) in
 let result = (match fdecl.A.typ with A.Void -> ""
                                            | _ -> f ^ "_result") in
     L.build_call fdef (Array.of_list actuals) result builder
  in

  (* Invoke "f builder" if the current block doesn't already
     have a terminal (e.g., a branch). *)
  let add_terminal builder f =
    match L.block_terminator (L.insertion_block builder) with
Some _ -> ()
    | None -> ignore (f builder) in

  (* Build the code for the given statement; return the builder for
     the statement's successor *)
  let rec stmt builder = function
A.Block sl -> List.fold_left stmt builder sl
    | A.Expr e -> ignore (expr builder e); builder
    | A.Return e -> ignore (match fdecl.A.typ with
  A.Void -> L.build_ret_void builder
| _ -> L.build_ret (expr builder e) builder); builder
    | A.If (predicate, then_stmt, else_stmt) ->
     let bool_val = expr builder predicate in
 let merge_bb = L.append_block context "merge" the_function in

```

```
  let then_bb = L.append_block context "then" the_function in
  add_terminal (stmt (L.builder_at_end context then_bb) then_stmt)
    (L.build_br merge_bb);

  let else_bb = L.append_block context "else" the_function in
  add_terminal (stmt (L.builder_at_end context else_bb) else_stmt)
    (L.build_br merge_bb);

  ignore (L.build_cond_br bool_val then_bb else_bb builder);
  L.builder_at_end context merge_bb

      | A.While (predicate, body) ->
   let pred_bb = L.append_block context "while" the_function in
   ignore (L.build_br pred_bb builder);

   let body_bb = L.append_block context "while_body" the_function in
   add_terminal (stmt (L.builder_at_end context body_bb) body)
     (L.build_br pred_bb);

   let pred_builder = L.builder_at_end context pred_bb in
   let bool_val = expr pred_builder predicate in

   let merge_bb = L.append_block context "merge" the_function in
   ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder);
   L.builder_at_end context merge_bb

     | A.For (e1, e2, e3, body) -> stmt builder
     ( A.Block [A.Expr e1 ; A.While (e2, A.Block [body ; A.Expr e3]) ] )
   in

   (* Build the code for each statement in the function *)
   let builder = stmt builder (A.Block fdecl.A.body) in

   (* Add a return if the last block falls off the end *)
   add_terminal builder (match fdecl.A.typ with
       A.Void -> L.build_ret_void
     | A.Int -> L.build_ret (L.const_int i32_t 0)
     | A.Float -> L.build_ret (L.const_float float_t 0.0)
     | A.Bool -> L.build_ret (L.const_int i1_t 0)
     | A.Complex -> L.build_ret (L.const_array float_t [|(L.const_float float_t 0.0);(L.const_float float_t 0.0)|] )
     | _ -> L.build_ret (L.const_int i32_t 0))


 in
```

```

  List.iter build_function_body functions;
  the_module
```

8.5 semant.ml

```
(*Semantic checking for the CompA compiler *)

open Ast
module StringMap = Map.Make(String)
(* module E = Exceptions *)


(* Semantic checking of a program. Returns void if successful,
   throws an exception if something is wrong.
   Check each global variable, then check each function *)
let check (globals, functions) =

  (* Raise an exception if the given list has a duplicate *)
  let report_duplicate exceptf list =
    let rec helper = function
  n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf n1))
      | _ :: t -> helper t
      | [] -> ()
    in helper (List.sort compare list)
  in

  (* Raise an exception if a given binding is to a void type *)
  let check_not_void exceptf = function
      (Void, n) -> raise (Failure (exceptf n))
    | _ -> ()
  in

  (* Raise an exception of the given rvalue type cannot be assigned to
     the given lvalue type *)
  let check_assign lvaluet rvaluet err =
     if lvaluet == rvaluet then lvaluet else raise err
  in

  let check_cxassign lvaluec index rvaluec err =
     if lvaluec == Complex && index== Int && rvaluec == Float then
    lvaluec else raise err
  in

```

```
(**** Checking Global Variables ****)

List.iter (check_not_void (fun n -> "illegal void global " ^ n))
 globals;

report_duplicate (fun n -> "duplicate global " ^ n) (List.map snd
 globals);

(**** Checking Functions ****)

if List.mem "print" (List.map (fun fd -> fd.fname) functions)
then raise (Failure ("function print may not be defined")) else ();

report_duplicate (fun n -> "duplicate function " ^ n)
  (List.map (fun fd -> fd.fname) functions);

(* Function declaration for a named function *)
let built_in_decls =  StringMap.add "print"
   { typ = Void; fname = "print"; formals = [( Float, "x")];
     locals = []; body = [] } (StringMap.add "sqrt"
   { typ = Float; fname = "sqrt"; formals = [(Float,"x")];
     locals = [];body =  []   } (StringMap.add "sin"
   { typ = Float; fname = "sin"; formals = [(Float,"x")];
     locals = [];body =  []   } (StringMap.add "cos"
   { typ = Float; fname = "cos"; formals = [(Float,"x")];
     locals = [];body =  []   } (StringMap.add "powi"
   { typ = Float; fname = "powi"; formals = [(Float,"x");(Int,"y")];
     locals = [];body =  []   } (StringMap.add "pow"
   { typ = Float; fname = "pow"; formals = [(Float,"x");(Float,"y")];
     locals = [];body =  []   } (StringMap.add "exp"
   { typ = Float; fname = "exp"; formals = [(Float,"x")];
     locals = [];body =  []   } (StringMap.add "log"
   { typ = Float; fname = "log"; formals = [(Float,"x")];
     locals = [];body =  []   } (StringMap.add "log10"
   { typ = Float; fname = "log10"; formals = [(Float,"x")];
     locals = [];body =  []   } (StringMap.add "fabs"
   { typ = Float; fname = "fabs"; formals = [(Float,"x")];
     locals = []; body =  []   } (StringMap.add "min"
   { typ = Float; fname = "min"; formals = [(Float,"x");(Float,"y")];
     locals = [];body =  []   } (StringMap.add "max"
   { typ = Float; fname = "max"; formals = [(Float,"x");(Float,"y")];
     locals = []; body =  []   } (StringMap.add "rnd"
   { typ = Float; fname = "rnd"; formals = [(Float,"x")];
     locals = []; body =  []   }(StringMap.singleton "printbig"
   { typ = Void; fname = "printbig"; formals = [(Int, "x")];
     locals = []; body = [] })))))))))))))
```

```
in


let function_decls = List.fold_left (fun m fd -> StringMap.add fd.fname fd m)
                         built_in_decls functions
in

let function_decl s = try StringMap.find s function_decls
     with Not_found -> raise (Failure ("unrecognized function " ^ s))
in

let _ = function_decl "main" in (* Ensure "main" is defined *)

let check_function func =

  List.iter (check_not_void (fun n -> "illegal void formal " ^ n ^
    " in " ^ func.fname)) func.formals;

  report_duplicate (fun n -> "duplicate formal " ^ n ^ " in " ^ func.fname)
    (List.map snd func.formals);

  List.iter (check_not_void (fun n -> "illegal void local " ^ n ^
    " in " ^ func.fname)) func.locals;

  report_duplicate (fun n -> "duplicate local " ^ n ^ " in " ^ func.fname)
    (List.map snd func.locals);

  (* Type of each variable (global, formal, or local *)
  let symbols = List.fold_left (fun m (t, n) -> StringMap.add n t m)
StringMap.empty (globals @ func.formals @ func.locals )
  in

  let type_of_identifier s =
    try StringMap.find s symbols
    with Not_found -> raise (Failure ("undeclared identifier " ^ s))
  in



  let matrix_access_type = function
      Matrix1DType(t, _) -> t
    | Matrix2DType(t, _, _) -> t
    | _ -> raise (Failure ("illegal matrix access") )
```

```
  in

  let check_pointer_type = function
      Matrix1DPointer(t) -> Matrix1DPointer(t)
    | Matrix2DPointer(t) -> Matrix2DPointer(t)
    | _ -> raise ( Failure ("cannot increment a non-pointer type") )
  in

  let check_matrix1D_pointer_type = function
    Matrix1DType(p, _) -> Matrix1DPointer(p)
    | _ -> raise ( Failure ("cannont reference non-1Dmatrix pointer type"))
  in

  let check_matrix2D_pointer_type = function
    Matrix2DType(p, _, _) -> Matrix2DPointer(p)
    | _ -> raise ( Failure ("cannont reference non-2Dmatrix pointer type"))
  in

  let pointer_type = function
  | Matrix1DPointer(t) -> t
  | Matrix2DPointer(t) -> t
  | _ -> raise ( Failure ("cannot dereference a non-pointer type")) in

  let matrix_type s = match (List.hd s) with
  | IntLit _ -> Matrix1DType(Int, List.length s)
  | FloatLit _ -> Matrix1DType(Float, List.length s)
  | BoolLit _ -> Matrix1DType(Bool, List.length s)
  | _ -> raise ( Failure ("Cannot instantiate a matrix of that type")) in

  let rec check_all_matrix_literal m ty idx =
    let length = List.length m in
      match (ty, List.nth m idx) with
    (Matrix1DType(Int, _), IntLit _) -> if idx == length - 1 then Matrix1DType(Int, length) else check_all_matrix_literal m (Matrix1DType(Int, length)) (succ idx)
  | (Matrix1DType(Float, _), FloatLit _) -> if idx == length - 1 then Matrix1DType(Float, length) else check_all_matrix_literal m (Matrix1DType(Float, length)) (succ idx)
  | (Matrix1DType(Bool, _), BoolLit _) -> if idx == length - 1 then Matrix1DType(Bool, length) else check_all_matrix_literal m (Matrix1DType(Bool, length)) (succ idx)
  | _ -> raise (Failure ("illegal matrix literal"))
```

```
in






  (* Return the type of an expression or throw an exception *)
  let rec expr = function
      IntLit _ -> Int
    | FloatLit _-> Float
    | StrLit _-> String
    | BoolLit _ -> Bool
    | Id s -> type_of_identifier s
    | ComplexAccess (s, e) -> let _ = (match (expr e) with
                                          Int -> Int
                                        | _ -> raise (Failure ("Complex index should be integer"))) in
                                         (type_of_identifier s)
    | Cx(e1,e2) -> let t1 =  expr e1 and t2 = expr e2 in
                    ( match t1 with
                      Float -> (match t2 with
                                  Float -> Complex
                                | _ -> raise (Failure ("illegal element type of Complex number " ^
                                        string_of_typ t2 ^" in " ^ string_of_expr e2)))
                    |_ -> raise (Failure ("illegal element type of Complex number " ^
                              string_of_typ t1 ^" in " ^ string_of_expr e1))
                    )

    | PointerIncrement(s) -> check_pointer_type (type_of_identifier s)
    | MatrixLiteral s -> check_all_matrix_literal s (matrix_type s) 0
    | Matrix1DAccess(s, e1) -> let _ = (match (expr e1) with
                                          Int -> Int
                                        | _ -> raise (Failure ("attempting to access with a non-integer type"))) in
                              matrix_access_type (type_of_identifier s)
    | Matrix2DAccess(s, e1, e2) -> let _ = (match (expr e1) with
                                          Int -> Int
                                        | _ -> raise (Failure ("attempting to access with a non-integer type")))
```

```
                                and _ = (match (expr e2) with
                                           Int -> Int
                                         | _ -> raise (Failure ("attempting to access with a non-integer type"))) in
                                matrix_access_type (type_of_identifier s)
    | Len(s) -> (match (type_of_identifier s) with
                  Matrix1DType(_, _) -> Int
                | _ -> raise(Failure ("cannot get the length of non-1d-matrix")))
    | Row(s) -> (match (type_of_identifier s) with
                  Matrix2DType(_, _, _) -> Int
                | _ -> raise(Failure ("cannot get the row of non-2d-matrix")))
    | Col(s) -> (match (type_of_identifier s) with
                  Matrix2DType(_, _, _) -> Int
                | _ -> raise(Failure ("cannot get the column of non-2d-matrix")))
    | Dereference(s) -> pointer_type (type_of_identifier s)
    | Matrix1DReference(s) -> check_matrix1D_pointer_type( type_of_identifier s )
    | Matrix2DReference(s) -> check_matrix2D_pointer_type( type_of_identifier s )

| Binop(e1, op, e2) as e -> let t1 = expr e1 and t2 = expr e2 in
(match op with
       Add | Sub | Mult | Div when t1 = Int && t2 = Int -> Int
| Add | Sub | Mult | Div when t1 = Complex && t2 = Complex -> Complex
| Add | Sub | Mult | Div when t1 = Float && t2 = Float -> Float
| Equal | Neq when t1 = t2 -> Bool
| Less | Leq | Greater | Geq when t1 = Int && t2 = Int -> Bool
| Less | Leq | Greater | Geq when t1 = Float && t2 = Float -> Bool
| And | Or when t1 = Bool && t2 = Bool -> Bool
      | _ -> raise (Failure ("illegal binary operator " ^
            string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
            string_of_typ t2 ^ " in " ^ string_of_expr e))
      )
    | Unop(op, e) as ex -> let t = expr e in
 (match op with
   Neg when t = Int -> Int
 | Not when t = Bool -> Bool
 | Neg when t = Complex -> Complex
 | Neg when t = Float -> Float
 | _ -> raise (Failure ("illegal unary operator " ^ string_of_uop op ^
         string_of_typ t ^ " in " ^ string_of_expr ex)))
```

```
  | Noexpr -> Void
  | Assign(e1, e2) as ex -> let lt = ( match e1 with
                                        | Matrix1DAccess(s, _) -> ( match (type_of_identifier s) with
                                            Matrix1DType(t, _) -> (match t with
                                                                  Int -> Int
                                                                | Float -> Float
                                                                | _ -> raise ( Failure ("illegal matrix of matrices") )
                                                              )
                                            | _ -> raise ( Failure ("cannot access a primitive") )
                                          )
                                        | Matrix2DAccess(s, _, _) -> (match (type_of_identifier s) with
                                            Matrix2DType(t, _, _) -> (match t with
                                                                  Int -> Int
                                                                | Float -> Float
                                                                | Matrix1DType(p, l) -> Matrix1DType(p, l)
                                                                | _ -> raise ( Failure ("illegal matrix of matrices") )
                                                              )
                                            | _ -> raise ( Failure ("cannot access a primitive") )
                                          )
                                        | _ -> expr e1)
                                   and rt = expr e2 in
  (*| Assign(var, e) as ex -> let lt = type_of_identifier var
                                   and rt = expr e in*)
      check_assign lt rt (Failure ("illegal assignment " ^ string_of_typ lt ^
            " = " ^ string_of_typ rt ^ " in " ^
            string_of_expr ex))
```

```
      | Cxassign(var,e1,e2) as ex -> let lt = type_of_identifier var
                                    and index = expr e1
                                    and num = expr e2 in
        check_cxassign lt index num (Failure ("illegal assignment of complex" ^ string_of_typ lt ^
             " = " ^ string_of_typ num ^ " in " ^
             string_of_expr ex  ^ "with" ^ string_of_typ index ))

      | Call(fname, actuals) as call -> let fd = function_decl fname in
         if List.length actuals != List.length fd.formals then
           raise (Failure ("expecting " ^ string_of_int
             (List.length fd.formals) ^ " arguments in " ^ string_of_expr call))
         else
           List.iter2 (fun (ft, _) e -> let et = expr e in
              ignore (check_assign ft et
                (Failure ("illegal actual argument found " ^ string_of_typ et ^
                " expected " ^ string_of_typ ft ^ " in " ^ string_of_expr e))))
             fd.formals actuals;
           fd.typ
    and

    check_bool_expr e = if expr e != Bool
     then raise (Failure ("expected Boolean expression in " ^ string_of_expr e))
     else ()



     (*match two ast*)
  and  expr_to_texpr e  = match e with
    IntLit(i)          -> Int
  | FloatLit(b)        -> Float
  | StrLit(s)          -> String
  | BoolLit(b)         -> Bool
  | Id(s)              -> expr e
  | Noexpr             -> Void
  | Unop(op, e)        -> expr e
  | Binop(e1, op, e2) as e   -> expr e
  | Call(_, el) as call        -> expr call
  | Cx(_,_)          -> Complex
in
  (* Library functions *)
```

```
(* and texpr_to_type texpr = match texpr with
    TIntLit(_, typ)                    -> typ
  | TFloatLit(_, typ)                  -> typ
  | TStrLit(_, typ)                    -> typ
  | TBoolLit(_, typ)                   -> typ
  | TId(_, typ)                        -> typ
  | TBinop(_, _, _, typ)               -> typ
  | TUnop(_, _, typ)                   -> typ
  | TCall(_, _, typ)                   -> typ
  | TCx(_,_,typ)                       -> typ
  | TNoexpr                            -> "void" in *)

    (* Verify a statement or throw an exception *)
  let rec stmt = function
  Block sl -> let rec check_block = function
           [Return _ as s] -> stmt s
         | Return _ :: _ -> raise (Failure "nothing may follow a return")
         | Block sl :: ss -> check_block (sl @ ss)
         | s :: ss -> stmt s ; check_block ss
         | [] -> ()
        in check_block sl
      | Expr e -> ignore (expr e)
      | Return e -> let t = expr e in if t = func.typ then () else
         raise (Failure ("return gives " ^ string_of_typ t ^ " expected " ^
                           string_of_typ func.typ ^ " in " ^ string_of_expr e))

      | If(p, b1, b2) -> check_bool_expr p; stmt b1; stmt b2
      | For(e1, e2, e3, st) -> ignore (expr e1); check_bool_expr e2;
                               ignore (expr e3); stmt st
      | While(p, s) -> check_bool_expr p; stmt s
    in

    stmt (Block func.body)

  in
  List.iter check_function functions
```

8.6 compa.ml

```
(* Top-level of the CompA compiler: scan & parse the input,
   check the resulting AST, generate LLVM IR, and dump the module *)
```

```
type action = Ast | LLVM_IR | Compile

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [ ("-a", Ast);      (* Print the AST only *)
                              ("-l", LLVM_IR);  (* Generate LLVM, don't check *)
                              ("-c", Compile) ] (* Generate, check LLVM IR *)
  else Compile in

  let file_to_string file =
    let array_string = ref [] in
      let ic = file in
          try
             while true do
                 array_string := List.append !array_string [input_line ic]
              done;
              String.concat "\n" !array_string
          with End_of_file -> close_in ic; String.concat "\n" !array_string

  in
  let in_file = open_in "stdlib.ca" in
  let string_in = file_to_string in_file in
  let other_file = file_to_string stdin in
  let str = String.concat "\n" [other_file; string_in] in


  let lexbuf = Lexing.from_string str  in
  let ast = Parser.program Scanner.token lexbuf  in
  (* Semant.check ast ; *)
  match action with
    Ast -> print_string (Ast.string_of_program ast)
   | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate ast))
   | Compile -> let m = Codegen.translate ast in
    Llvm_analysis.assert_valid_module m;
    print_string (Llvm.string_of_llmodule m)
```

8.7 exceptions.ml

```
exception UnsupportedMatrixType

exception IllegalAssignment

exception IllegalPointerType

exception MatrixOutOfBounds

exception IllegalUnop

exception WrongReturn
```

8.8 makefile

```
# Make sure ocamlbuild can find opam-managed packages: first run
#
# eval `opam config env`

# Easiest way to build: using ocamlbuild, which in turn uses ocamlfind

.PHONY : all
all : compa.native

.PHONY : compa.native
compa.native :
	ocamlbuild -use-ocamlfind -pkgs llvm,llvm.analysis -cflags -w,+a-4 \
		compa.native

# "make clean" removes all generated files

.PHONY : clean
clean :
	ocamlbuild -clean
	rm -rf testall.log *.diff compa scanner.ml parser.ml parser.mli
	rm -rf *.cmx *.cmi *.cmo *.cmx *.o *.s *.ll *.out *.exe

# More detailed: build using ocamlc/ocamlopt + ocamlfind to locate LLVM

OBJS = ast.cmx codegen.cmx parser.cmx scanner.cmx semant.cmx compa.cmx

microc : $(OBJS)
	ocamlfind ocamlopt -linkpkg -package llvm -package llvm.analysis $(OBJS) -o compa

```

```
scanner.ml : scanner.mll
	ocamllex scanner.mll

parser.ml parser.mli : parser.mly
	ocamlyacc parser.mly

%.cmo : %.ml
	ocamlc -c $<

%.cmi : %.mli
	ocamlc -c $<

%.cmx : %.ml
	ocamlfind ocamlopt -c -package llvm $<

### Generated by "ocamldep *.ml *.mli" after building scanner.ml and parser.ml
ast.cmo :
ast.cmx :
codegen.cmo : ast.cmo
codegen.cmx : ast.cmx
compa.cmo : semant.cmo scanner.cmo parser.cmi codegen.cmo ast.cmo
compa.cmx : semant.cmx scanner.cmx parser.cmx codegen.cmx ast.cmx
parser.cmo : ast.cmo parser.cmi
parser.cmx : ast.cmx parser.cmi
scanner.cmo : parser.cmi
scanner.cmx : parser.cmx
semant.cmo : ast.cmo
semant.cmx : ast.cmx
parser.cmi : ast.cmo

# Building the tarball

TESTS = hello

TESTFILES = $(TESTS:%=test-%.mc) $(TESTS:%=test-%.out) \
      $(FAILS:%=fail-%.mc) $(FAILS:%=fail-%.err)

TARFILES = ast.ml codegen.ml Makefile _tags microc.ml parser.mly README \
        scanner.mll semant.ml testall.sh printbig.c arcade-font.pbm font2c \
	$(TESTFILES:%=tests/%)

microc-llvm.tar.gz : $(TARFILES)
	cd .. && tar czf microc-llvm/microc-llvm.tar.gz \
```

```
    $(TARFILES:%=microc-llvm/%)
```

8.9 demo1.ca

```
/* A simple complex number operation problem
   Scenario: A user wants to compute a expression
   Given z1 = 2.0 + 3.1i, z2 = 10.3 + 23.1i, z3 = 1.2, z4 = i
   Question: Get the euler form of (z1*z2 + z1/(z2-z4))/z3 */


int main(){
    cx z1;
    cx z2;
    cx z3;
    cx z4;
    cx euler_form;

    z1 = (2.0, 3.1);
    z2 = (2.3, 2.1);
    z3 = (1.2, 2.2);
    z4 = (0.0, 4.0);


    euler_form = euler(div_complex( add_complex(mult_complex(z1,z2), div_complex(z1,sub_complex(z2,z4))), z3));
    println(euler_form);
}
```

8.10 demo2.ca

```
/* An example of users verifying that properties of complex number holds
   Suppose an user is not sure some properties in complex numbers
   He or she can verify these properties by writing own programs to check
   For example: Magnitude of z and its conjugate should be the same */


int main(){
    cx a;
    bool check;
    a = (2.0,3.0);
```

```

    check = conj_check(a);
    println("check if magnitudes of z and its conjugate are the same:");
    if (check == true){
        println("correct");
    } else {
        println("incorrect");
    }
}


/*
    User-defined function
*/

/* check if magnitudes of z and its conjugate are the same */
bool conj_check(cx a){
    float b;
    float c;
    cx conj_a;

    conj_a = conj_complex(a);
    b = mag_complex(a);
    c = mag_complex(conj_a);
    if (b == c){
        return true;
    } else {
        return false;
    }
}
```

8.11 demo3.ca

```
int main()
{
    float[2][3] m1;
    float[2][3] m2;

    populate_2D_int(%%m1, 1.0, row(m1), col(m1));
    populate_2D_int(%%m2, 2.0, row(m2), col(m2));

    add_2D_scalar(%%m1, 5.0, row(m1), col(m1));
    add_2D_int(%%m1, %%m2, row(m1), col(m1));
```

```

    print_2D_int(%%m1, row(m1), col(m1));
}

void populate_2D_int(float[][] x, float a, int r, int c) {
    int i;
    for (i=0;i<(r*c);i=i+1) {
        #x = a;
        x = ++x;
    }
}

void print_2D_int(float[][] x, int r, int c) {
    int i;
    int j;

    for (i=0; i<r; i=i+1) {
        print("[  ");
        for (j=0; j<c; j=j+1) {
            print(#x);
            print("   ");
            x = ++x;
        }
        println("]");
    }
}

void add_2D_scalar(float[][] x, float scalar, int r, int c) {

    int i;

    for (i=0; i<(r*c); i=i+1) {
        #x = #x + scalar;
        x = ++x;
    }
}


void add_2D_int(float[][] x, float[][] y, int r, int c) {

    int i;

    for (i=0; i<(r*c); i=i+1) {
        #x = #x + #y;
        x = ++x;
        y = ++y;
```

```
    }
}
```

8.12 hello.ca

```
int main()
{
    float a;
    cx b;

    b =(0.0,0.0);
    a = sin(60.3);
    println(a);
    a = cos(32.4);
    println(a);
    a = exp(32.4);
    println(a);
    a = powi(32.4,3);
    println(a);
    a = powi(32.4,3);
    println(a);
    b[0]=pow(32.4,3.0);
    b[1]=min(32.1,34.5);
    println(b);
    b[0] =fabs(-21.3);
    b[1] = max(23.4,23.3);
    println(b);
    b[0] =log(21.3);
    b[1] = log10(23.4);
    println(b);
    return 0;
}
```

8.13 stdlib.ca

```
/* Euler function */
cx euler(cx c){
    float a1;
    float a2;
    float a3;
    int i;
    cx result;
```

```

    result = (0.0,0.0);
    a1 = sin(c[1]);
    a2 = cos(c[1]);
    a3 = pow(exp(1.0), c[0]);

    result[0]= a3*a1;
    result[1]= a3*a2;
    return result;
}


/* Complex addition */
cx add_complex(cx a, cx b){
    cx result;
    result = (0.0,0.0);
    result[0] = a[0]+b[0];
    result[1] = a[1]+b[1];
    return result;
}


/* Complex subtraction */
cx sub_complex(cx a, cx b){
    cx result;
    result = (0.0,0.0);
    result[0] = a[0]-b[0];
    result[1] = a[1]-b[1];
    return result;
}


/* Complex multiplication */
cx mult_complex(cx a, cx b){
    cx result;
    result = (0.0,0.0);
    result[0] = a[0]*b[0]-a[1]*b[1];
    result[1] = a[0]*b[1]-a[1]*b[0];
    return result;
}


/* Complex division */
cx div_complex(cx a, cx b){
    cx result;
    result = (0.0,0.0);
```

```
    result[0] = (a[0]*b[0] + a[1]*b[1]) / (b[0]*b[0] + b[1]*b[1]);
    result[1] = (a[1]*b[0] - a[0]*b[1]) / (b[0]*b[0] + b[1]*b[1]);
    return result;
}


/* Complex power */
cx pow_complex(cx a, int n){
    cx result;
    int i;
    result = a;

    for (i = 0; i<(n-1); i=i+1){
        result =  mult_complex(result,a);
    }
    return result;
}


/* Complex magnitude */
float mag_complex(cx a){
    float result;
    result = sqrt(a[0]*a[0]+a[1]*a[1]);
    return result;
}


/* Complex conjugate */
cx conj_complex(cx a){
    cx result;
    float temp;
    temp = a[1];
    result = (0.0,0.0);
    result[0] = a[0];
    result[1] = 0.0-temp;
    return result;
}
```