COMSW4115 Course Project

EasyCircuit

Final Report

Lei Zhang |LZ2343 **Xingyue Zhou |XZ2299**
Liming Zhang |LZ2342 **Yingnan Li|YL2884**
Wei Zhang |WZ2254

Introduction

Circuit design plays an important role in many areas. However, some circuit operations, such as computing equivalent resistance, or transforming a circuit, though intuitionally in some ways, are still hard to formularily express in mathematics or programs. Famous electronic design automation (EDA) and simulation program, such as Multisiml, can help engineers to solve these problems. However, for entry-level students, these IDEs seem to be complex and expensive. To overcome such weakness, we want to design a light-weight, succinct, accurate and easy-to-control programming language, called Easy Circuit, to facilitate people in designing, controlling and analyzing circuits. By using EasyCircuit, people whom as long as possess basic knowledge of programming and circuit, can design a circuit, model it into a component with a function defining its properties, and then do some relative operations and computation as you wish. Such modularized components can be reused in a more complicated electronic system. So, EasyCircuit also has a great prospect.

Project Overview

Easy Circuit is a special language for people to design circuit in several aspects as followed:

✧ In our Easy Circuit language, user can conveniently define their own electronic components, such as combination of certain resistor and

computation of bits, by using the data type and operator in our language.

- ✧ Easy Circuit language can implement not only the basic operators, but also particular operators especially for circuit, such as series connection, parallel connection, and reciprocal. And for bits, there are 'and' and 'or' operator available.
- ✧ The user may define a function to describe the property of such template, where the variant electronic components corresponds to the parameters in the function, and the function maps these variants to the output according to the user's definition of the function.

Goal of EasyCircuit

- ✧ **Easy to learn**: The language style is similar to Java, and contains basic primitive types and operators, such as int, float, string, +, -, */, etc.
- ✧ **Have customized operators and data structures**: Our language contains operators and types especially for circuit design and calculation, such as series connection, parallel connection, and reciprocal; resistance, wire, bit, etc.
- ✧ **Functional:** User can use EasyCircuit to model a circuit, i.e., to define a function to describe the properties of certain circuit. In functions, the parameters are the adjustable component in the corresponding circuit.

Language Tutorial

Get Started

The EasyCircuit compiler can generate intermediate Java source code without the Java compiler installed. In order for the EasyCircuit compiler to create a runnable program, a Java compiler of at least version 1.6 needs to be available in the path of the user compiling the Java program. The Java compiler is packaged with the Java Development Kit (JDK). To run a compiled EC program, the Java Run-time Environment (JRE) must be installed.

Installing the compiler

- **Under Windows7**

1. Go to the directory where our compiler located.

2. double clik make. bat or make use of Makefile.

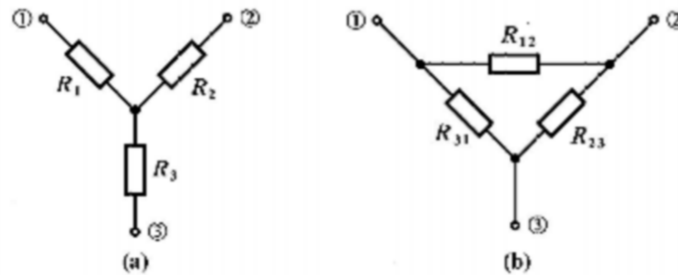
- **Under Mac osX**

1. Modify the last line ec.exe->ec as follows in make.bat file.

ocamlc -o ec ast.cmo translator.cmo parser.cmo scanner.cmo ec.cmo

2. In **TERMINAL** in Mac ios, go to the directory where our compiler located.
3. Type". /make.bat" in the commend line.

How to write Your First Example: Y-Configuration and Delta Configuration Example



y configuration(left) and delta configuration(right)

1. how to write a main function

EC language must have a main function. This function servers as a main entry point into the program, as shown below.

```
void main()
{
//main function body
}
```

Or we can define the type of return of main function like:

```
bit main(){
bit input;
input.value=”1000000”b;
return input;
}
```

We can also find how to declare, define a bit type (our special data type) in this example code.

2. Get to know our Basic Value

✧ how to declare a variable, define value and scope.

In our EC language, u must separate the declaration and the assignment parts.

```
res r1;
res r2;
res r3;
r1.value=1;
//the scope of r1 is the expressions after this line, before the function ends.
r2.value=2;
r3.value=3;
```

3. How to use the build-in function.

We have one build-in function: **show** function, which can be used in those ways below:

```
show(((r1#r2)$r3).value);//show the computed value
show(2+4);//show the computed result
show("this is show test");//show the string
show(add(6,9));//add is a function we have defined
```

4. How to declare and define functions

Function declaration and definition looks similar to C style. Scoping works in a similar way, through function's scope begins immediately after the arguments,

```
res equi_res(res r1, res r2, res r3)
{
res rtotal;
rtotal=(((r1$r2)#r3)#r4);
return rtotal;
}
```

5. Use the operations defined by our EC languge.

```
r12.value=(r1.value*r2.value+r2.value*r3.value+r3.value*r1.value)/r3.value;
r23.value=(r1.value*r2.value+r2.value*r3.value+r3.value*r1.value)/r1.value;
r13.value=(r1.value*r2.value+r2.value*r3.value+r3.value*r1.value)/r2.value;
r4=(r1#r2)$r3
```

6. Write an example for Circuit transformation-From Y-Registor to Star-Registor

You can create a txt file and write a short code like this and change the name of postfix ".txt" to ".ec". And then, you can save it in the Project Folder "ec_test_case".

```
void main(){
res r1;
res r2;
res r3;
res r12;
res r13;
res r23;
r1.value=1;
r2.value=2;
r3.value=3;
r12.value=(r1.value*r2.value+r2.value*r3.value+r3.value*r1.val
ue)/r3.value;
r23.value=(r1.value*r2.value+r2.value*r3.value+r3.value*r1.val
ue)/r1.value;
r13.value=(r1.value*r2.value+r2.value*r3.value+r3.value*r1.val
ue)/r2.value;

show(r12.value);
show(r13.value);
show(r23.value);
}
```

7. Compilation and execution our .ec file

✧ Under Windows 7

Double click test_ec.bat

✧ Under Mac ios

Double click test_ec.sh

At the same time, we can get the Java source code in the corresponding .java code in the Project Folder.

Additional Example #1: gcd Function

```
void main()
{
  show(gcd(2,14));
  show(gcd(3,15));
  show(gcd(99,121));
   }
int gcd(int a, int b)
{
  while (a != b) {
    if (a > b)
    {
       a = a - b;
    }
    else
    {
       b = b - a;
    }
  }
  return a;
```

Additional Example #2: Chip Function

```
bit main(){
bit input;
bit output;
input.value= "100000"b;
return output=chip1(input);
}
bit chip1(bit input){
bit output;
if (input.value == "100000"b)   output.value="01111111"b;
else if  (input.value == "100001"b)   output.value="10111111"b;
else if  (input.value == "100010"b)   output.value="11011111"b;
else if  (input.value == "100011"b)   output.value="11101111"b;
else if  (input.value == "100100"b)   output.value="11110111"b;
else if  (input.value == "100101"b)   output.value="11111011"b;
else if  (input.value == "100110"b)   output.value="11111101"b;
else if  (input.value == "100111"b)   output.value="11111110"b;
else output.value="input error.";
show("Input is :");
show(input.value);
show("Output is :");
show(output.value);
return output;

}
```

Language Reference Manual

Lexical conventions

Tokens

There are 5 classes of tokens: identifiers, keywords, whitespace, operators, and other separators.

Comments

The characters /* introduce a comment, which is terminated by the characters */.
The characters // also introduce a comment, which is terminated by the newline character. Comments do not nest. Lines marked as comments are discarded by the compiler.

Whitespace

Blanks, tabs, and newlines, collectively referred to as "white space" are ignored except to separate tokens.

Identifiers

An identifier is a sequence of letters and digits, the first character must be alphabetic. The underscore "_ " counts as alphabetic. Upper and lower case letters are considered different.

Keywords

The following identifiers are reserved as keywords and may not be used otherwise. In this manual, keywords are bolded.

- if
- else
- for
- while
- return
- int
- float
- new
- res
- bit
- function
- default
- value
- void
- boolean

Operators

An operator is a token that specifies an operation on at least one operand. The operand may be an expression or a constant.

Arithmetic operators:+,-, *, /, %, ~

Comparison operators:<,>,<=,>=, !=, ==

Bitwise operators: &, |

Assignment operator:=

Circuit operator: # $

Punctuation

Parenthesis: “()”
Brace: “{}”
Bracket: “[]”
Semicolon: “;”
Comma: “:”
Dot: “.”

Expression and Operators

Easy Circuit language is described using expressions which are made up of one or more operators and operands.

Assignment

There are two special types need to be mentioned in this part, **res** and **bit.**
res refers to the type of resistance, which takes in decimal value.
bit refers to the type that describes the chips and modules in a digital circuit, which takes in binary strings.
Note that declaration is a must before assignment.

(a)res assignment

```
res resistance_name   //declaration
resistance_name.value= resistance_value; //assignment
```

example:

```
res r1;
```

```
r1.value=6;          //assigning a decimal value
```

(b) bit assignment

```
bit chip_name;  //declaration
chip_name.value="chip_value"; //assignment
```

example:

```
bit b1;
b1.value= "001101";           //assigning a binary value
```

Series Connection, Parallel Connection

To place two or more resistance in series or in parallel connection, we use the concatenation syntax and parenthesis to represent the highest precedence.

(a) Series:

```
res r_series = r1#r2;
```

(b) Parallel:

```
res r_parallel = r1$r2;
```

(c) Combination of series and parallel connection in left-right order.

```
res rc = r1#r2$r3#r4;
// result in: res rc = r1#r2;  rc = rc$r3;  rc = rc#r4;
```

(d) Parenthesis

Parenthesis has the highest precedence.

```
(r1#r2)$r3;      //result in r1#r2$r3
r1#(r2$r3);      //result in r2$r3#r1
```

Arithmetic

Arithmetic operators are shorthand for common equal operations. Most of they operate on the value of two resistances or their equal resistances. An exception is that Reciprocal operation is on the value of one resistance or its equivalent resistance. All operations are done in two's complement and they will `return` the value of the new resistance.

```
res r1;
res r2;
res r3;
res r4;
res r5;

r1.value=1;
r2.value=2;
r4.value=1.0;
r5.value=2.0;
r3.value=r1.value+r2.value;
```

Operator	Example	Result	Note
Plus	`r3.value=r1.value+r2.value`	`r3.value = 3`	
Minus	`r3.value=r2.value-r1.value`	`r3.value = 1`	`r.value should be more than or equal to zero.`
Multiplication	`r3.value=r1.value*r2.value`	`r3.value = 2`	

Division	r3.value=r1.value/r2.value r3.value=r4.value/r5.value	r3.value = 0 r3.value=0.5	The data type of r.value depends on the data type of its divisor and dividend.
Reciprocal	r3.value = ~r5.value r3.value = ~r2.value	r3.value=0.5 r3.value = 0	It returns the reciprocal value. The data type depends on the original data type.

Binary

Bit operators represents the basic **AND**, **OR** operation.

```
res a;
res b;
a.value="01";
b.value="11";
```

Operator	Example	Result
AND	a&b	"01"
OR	a|b	"11"

Comparison

Comparison operators will compare the values of two resistances which do not to be equally sized and will return a Boolean type value.

```
res r1;
res r2;

r1.value=1;
r2.value=2;
```

Operator	Example	Result
Less than	r1.value < r2.value	True
Less than or equal to	r1.value <= r2.value	True
Greater than	r1.value > r2.value	False
Greater than or equal to	r1.value >= r2.value	False
Equal to	r1.value == r2.value	False
Not equal to	r1.value != r2.value	True

Precedence and associativity

Operators, in order of decreasing precedence	Associativity
++ -- (postfix, for-loops only)	Left to right
+ - ~(unary) ++ -- (prefix, for-loops only)	Right to left
* /	Left to right
+ -(binary)	Left to right
<<= >>=	Left to right
== !=	Left to right
& (binary)	Left to right
| (binary)	Left to right
# $ (binary)	Left to right
=	Right to left

Declaration

A declaration is a must before assignment. The form of declaration is similar to that of C language.

Identifiers

Identifiers are user-friendly names, much like variable names in most programming languages. They must start with upper-or lower-case letter followed by any sequence of letters, numbers, and underscores. No other characters are allowed in identifier names.

Resistance Declaration

Resistances are declared using the keyword res:

```
res resistance_name;
```

Chip Declaration

Chips and modules are declared using the keyword bit:

```
bit chip_name;
```

Function Declaration

Easy Circuit language support user-defined and system-defined functions. Normally user of Easy Circuit can use functions to model a circuit component with specific properties (e.g., to compute equivalentresistanceof such circuit component) and to reuse it in a more complicated electronic system.
Like C, a function in Easy Circuit contains zero or more statements to be executed when it is called, can be passed zero or more arguments, and can return a value.

Function types

A function's type means the type of the value that is returned after executing the function. The type can be any data type in Easy Circuit Language. If the function returns no value, the type of function is void.

Function definitions

In Easy Circuit Language, a function has the following syntax:

```
<function-type><function-name> (parameter1-type parameter1-name,
parameter2-type parameter2-name,……)
{
statement 1;
statement 2;
…
return statement;
}
```

In the function definition, the function uses the input parameters to execute the statements and return the value of the statement, with the type defined in`<function-type>`

For example, the following function equi_res4can compute equivalent resistance of 4 parallel connected resistors.

```
res equi_res4 (res r1, res r2, res r3, res r4)
{
Res res_equi = r1$r2$r3$r4;
Return res_equi;
}
```

Function Calls

A function call is a primary expression. Just like in C or Java, in Easy Circuit Language, an expression used to invoke a function has the following format:

```
<function-name> (parameter1-name, parameter2-name,……);
```

`(parameter1-name, parameter2-name,……)` contains a comma-separated list of expressions that are the arguments to the function. The following is an example of a call to the function equi_res4 defined in the previous part.

```
{
res r1;
res r2;
res r3;
res r4;
res r_total;

r1.value=2;
r2.value=2;
r3.value=2;
r4.value=2;
r_total =equi_res4 (r1,r2,r3,r4);
}
```

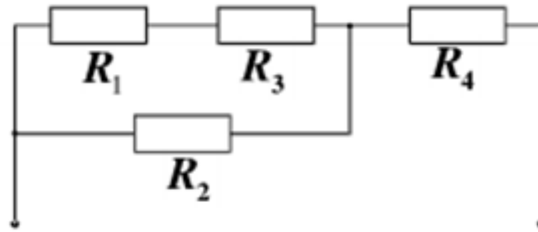
After execute the code above the r_total will be assigned a res type variant with value of 0.5.

Function Scope

In Easy Circuit Language, variant declared within one function is valid only in that function. The functions that have been defined will be accessible in the whole file.

Example

In the following code, a function called get_equal_res was defined to module the circuit component in the figure.




```
res get_equal_res (res r1, res r2, res r3, res r4)
{
res res_total;
res_total.value=(((r1#r3)$r2)#r4).value;
return res_total;
}
// Define the function to calculate the equivalentresistance in
circuit module showed in the picture.

res r1; // Assign the values of R1, R2, R3 and R4
res r2;
res r3;
res r4;
res rTotal;

r1.value=3;
r2.value=6;
r3.value=3;
r4.value=3;
rTotal.value= get_equal_res(r1, r2, r3, r4);
// Get the equivalentresistance.
```

Statements

A semicolon is necessary after a statement in Easy Circuit. Because whitespace has no effect, the semicolon is used to signal the end of a statement.

```
Res rTotal;
rTotal.value=13;
```

Conditional Statement

Conditional statements work just as they do in C with if and else.

```
If (r1.value < r2.value){
r3= r1;
    }
Else {
r4= r2;
    }
```

For-loops

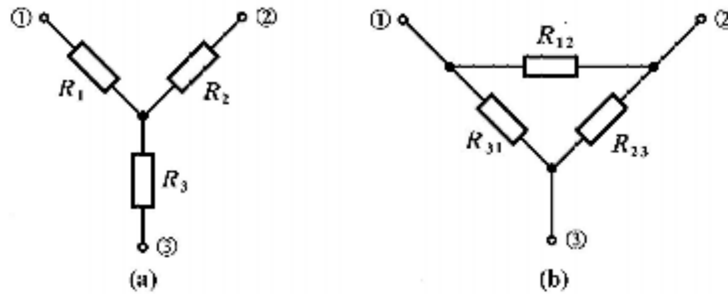
For loops in Easy Circuit is similar to the for loops in C

```
For(i=0; i<18; i++){
   …….;
   }
```

Sample code

Circuit transformation

As the figure shows as follow, y configuration and delta configuration are commonly used in circuit calculation. They can be transformed into each are by the formulas show as following.



y configuration(left) and delta configuration(right)

$$
\begin{cases}
R_{12} = \dfrac{R_1R_2 + R_2R_3 + R_3R_1}{R_3} \\
R_{23} = \dfrac{R_1R_2 + R_2R_3 + R_3R_1}{R_1} \\
R_{31} = \dfrac{R_1R_2 + R_2R_3 + R_3R_1}{R_2}
\end{cases}
\qquad
\begin{cases}
R_1 = \dfrac{R_{12}R_{31}}{R_{12} + R_{23} + R_{31}} \\
R_2 = \dfrac{R_{12}R_{23}}{R_{12} + R_{23} + R_{31}} \\
R_3 = \dfrac{R_{23}R_{31}}{R_{12} + R_{23} + R_{31}}
\end{cases}
$$

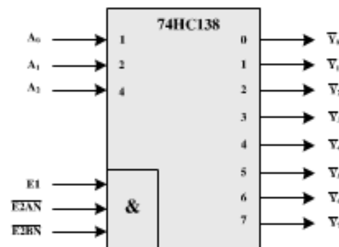
Formula of transformation between y configuration and delta configuration

The following code transforms the y configuration to delta configuration:

```
void main(){
res r1;
res r2;
res r3;
res r12;
res r13;
res r23;
r1.value=1;
r2.value=2;
r3.value=3;
r12.value=(r1.value*r2.value+r2.value*r3.value+r3.value*r1.value)
/r3.value;
r23.value=(r1.value*r2.value+r2.value*r3.value+r3.value*r1.value)
/r1.value;
r13.value=(r1.value*r2.value+r2.value*r3.value+r3.value*r1.value)
/r2.value;
show(r12.value);
show(r13.value);
show(r23.value);}
```

74HC138 3-to-8 line decoder

E1	E2AN	E2BN	A_2	A_1	A_0	$\overline{Y_0}$	$\overline{Y_1}$	$\overline{Y_2}$	$\overline{Y_3}$	$\overline{Y_4}$	$\overline{Y_5}$	$\overline{Y_6}$	$\overline{Y_7}$
X	1	X	X	X	X	1	1	1	1	1	1	1	1
X	X	1	X	X	X	1	1	1	1	1	1	1	1
0	X	X	X	X	X	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	1	1	1	1	1	1	1
1	0	0	0	0	1	1	0	1	1	1	1	1	1
1	0	0	0	1	0	1	1	0	1	1	1	1	1
1	0	0	0	1	1	1	1	1	0	1	1	1	1
1	0	0	1	0	0	1	1	1	1	0	1	1	1
1	0	0	1	0	1	1	1	1	1	1	0	1	1
1	0	0	1	1	0	1	1	1	1	1	1	0	1
1	0	0	1	1	1	1	1	1	1	1	1	1	0



The following code implements the 74HC138 3-to-8 line decoder,

```
void main(){
bit input;
bit output;
input.value="1000000";
output=74HC138 (input);
}
bit 74HC138 (bit input){
bit output;
if  (input.value   == "100000")   output.value="01111111";
else if  (input.value == "100001")   output.value="10111111";
else if  (input.value == "100010")   output.value="11011111";
else if  (input.value == "100011")   output.value="11101111";
else if  (input.value == "100100")   output.value="11110111";
else if  (input.value == "100101")   output.value="11111011";
else if  (input.value == "100110")   output.value="11111101";
else if  (input.value == "100111")   output.value="11111110";
else show("input error.");
return output;
}
```

gcd

The following code does the simply "gcd" function in our language

```
void main()
{
  show(gcd(2,14));
   show(gcd(3,15));
   show(gcd(99,121));
   }

int gcd(int a, int b)
{
    while (a != b) {

    if (a > b)
    {
       a = a - b;
    }
    else
    {
       b = b - a;
    }
  }
  return a;
}
```

Project Plan

Team Responsibility

Scanner & Parser& AST	LimingZhang Group Discussed
Semantics AST	Liming Zhang, WeiZhang
Translator	Xingyuezhou, LeiZhang
Test	Xingyuezhou, Yingnan Li
Final Report	Yingnan Li Group Discussed

Architecture Design

Overview

Type check Design and Implementation

After finishing lexical and syntax analysis part, we come to consider semantics analysis. We want to make sure the code of our language has correct data type, appropriate scope, right usage of operands and no re-declaration or un-declaration of variables and functions. Therefore, in the process of semantics analysis, we need to deal with type check and symbol table.

As what mentioned above, first we should take into consideration the specific data type in our language: res and bit type. When declaring a res or bit type, we regulate that only int, float or res type can be used for a resistance and only bit or string can be used for a binary circuit since they have particular meaning in the circuit language.

Moreover, regarding special operands in our language, we should also check the type to ensure the appropriate data type using the operands. For instance, when taking advantage of series "#" or parallel "$", we need to check that the data type must be res since it is only meaningful for resistance in our language. Similarly, reciprocal "~" can be only used for an int, float or res type, which is commonly calculated when dealing with circuit.

Meanwhile, we should also account for correctly calling a function or declaring a variable. First we will search local scope to see whether the function or the variable has been defined or not. Then we continue to search them in the global scope with the help of symbol table. In this way, we can find whether they are undeclared or re-declared, which is not allowed in our language. And our EC also designs a built-in function "show". We regarded it as already defined and will not raise a failure for this built-in function.

The type checking part requires thorough consideration and rigors implementation. In order to make our language precise and elaborate, the type checking part must be designed to be comprehensive. There are many things we need to consider, for example, how to define and maintain scope, whether the function allows implicit conversion, whether the break is allowed in in for loop, and how to deal with build-in function. More detailed considered, if the bool type can be used in arithmetic as 0 or 1, how to figure out life cycle for a variable, etc.
After semantic analysis, we will store all the information in the SAST(semantically abstract syntax tree), which is similar to AST but is determined with checked and appropriate data type and functions.

Design and Implementation of Translator

After building up the sematic abstract syntax tree (SAST), we started to write the translator part, whose function is to generate Java sources code based on the SAST of our Easy Circuit language.
The overall translation scheme is to translate variables and functions complying with Java grammar, which mainly consists of translating variable declarations, types, function declarations, statements, and expressions. All of them is described in one function in OCaml and is used mutually and recursively with itself and each other.
As stated in the previous parts, the feature of our language is its special data type of resistances and binary expressions, as well as two special operators of parallel connection and series connection. Regarding to the implementation of our special data type, we took advantage of the Java inner class. That is we create an inner class called Res in Java when dealing

with res type in our Easy Circuit language. Implementation for type bit is similar to type res.

In order to generate succinct Java source code, at the beginning of the code translation, we need to determine if there exists an res(or bit, or both) type in the Easy Circuit language. If there exists an res(bit) type, then we would need to generate the Java code for Res(Bit) inner class. Otherwise, we do not. For example, in the implementation of gcd function, we do not need the inner class definition of either Res or Bit, thus we do not generate the related Java source code for such type.

To implement this, at the beginning part of translating procedures, we go over all the elements in the variable type list to see if there exist res or bit type or none of them, and then generate the corresponding Java code at the beginning of the Java source code.

Based on the strategy mentioned above, we can generate the "header" of our target Java source code. As the Ocaml source code shows as follow:

```
let vars_type_list = global_vars_type_list
@(funcs_locals_type_list)@funcs_formals_type_list
  in
  if( List.mem  Ast.Res vars_type_list && List.mem  Ast.Bit vars_type_list) then begin_code_with_res_and_bit
  else if( List.mem  Ast.Res vars_type_list) then begin_code_with_res
  else if( List.mem  Ast.Bit vars_type_list) then begin_code_with_bit
  else begin_code_necessary

let jstring_of_program (vars, funcs)  file_name  =
  let header=jstring_of_header (vars, funcs)in
  let global_vars = List.map (fun a -> { vvname = a.vvname; vvtype = a.vvtype }) vars
  in
  package_del ^ "\n" ^ import_decl ^ "\n\n" ^
  "public class " ^ (String.sub file_name 0 ((String.length file_name) - 3)) ^"_result"^
  "\n{\n" ^header  ^
  String.concat "" (List.map jstring_of_gvdecl vars) ^ "\n\n" ^
  String.concat "\n" (List.map (jstring_of_fdecl global_vars) funcs) ^
  "\n}"
```

For our special operators "#" and "$", we generator Java function to compute the equivalent resistance of parallel connected resistance and

serially connected resistance.

To translate Easy Circuit type into Java code is straight forward, as the following code shows:

```
let jstring_of_datatype dtype =
  match dtype with
    Ast.Int -> "int"
  |  Ast.Float -> "double"
  |  Ast.Boolean -> "boolean"
  |  Ast.Void -> "void"
  | Ast.String -> "String"
  | Ast.Res -> "Res"
  |  Ast.Bit ->"Bit"
```

To translate Easy Circuit expression into Java code, firstly we match the different cases. If it is a literal, then it is simple, and we can just print them straightforward. If it is a binary operation then translate the two expressions on both side of the operator recursively and connect them with printing the operator. Same procedure is used when dealing with Unary, After op, Assign and Call.

Based on translating expressions and other functions defined before, the function to translate statement is implemented as the following code shows:

```
let rec jstring_of_stmt global_vars local_vars = function
  Block(decls, stmts) ->
    "{\n" ^ String.concat "" (List.map (jstring_of_stmt global_vars local_vars) stmts) ^ "}\n"
| Expr(exprt) -> let (expr,expru)=exprt in jstring_of_expr global_vars local_vars expr ^ ";\n";
| Return(exprt) -> let (expr,expru)=exprt in  "return " ^ jstring_of_expr global_vars local_vars expr ^ ";\n";
| If(et, s, Block([],[])) ->  let (e,eu)=et in "if (" ^ jstring_of_expr global_vars local_vars e ^ ")\n" ^ jstring_of_stmt global_vars local_vars s
| If(et, s1, s2) -> let (e,eu)=et in "if (" ^ jstring_of_expr global_vars local_vars e ^ ")\n" ^
    jstring_of_stmt global_vars local_vars s1 ^ "else\n" ^ jstring_of_stmt global_vars local_vars s2
| For(e1t, e2t, e3t, s) -> let (e1,e1u)=e1t in let (e2,e2u)=e2t in let (e3,e3u)=e3t in
    "for (" ^ jstring_of_expr global_vars local_vars e1  ^ " ; " ^ jstring_of_expr global_vars local_vars e2 ^ " ; " ^
    jstring_of_expr global_vars local_vars e3  ^ ") " ^ jstring_of_stmt global_vars local_vars s
| While(et, s) ->  let (e,eu)=et in "while (" ^ jstring_of_expr global_vars local_vars e ^ ") " ^ jstring_of_stmt global_vars local_vars s
```

Test Plan

Our test plan consists of pairs of identically named files in the **ec_test_case**. Each pair consists of a EasyCircuit program with extension **.ec** and **.java file.** The **.ec** file contains the code to be tested,

Our testing is divided into four sections: **syntax verification, semantic/type verification**, **data type verification**

Syntax Verification

Syntax verification testing is meant to confirm that the parser accepts all valid token strings, and rejects all invalid ones as defined in our language reference manual.

Semantic Verification

Semantic verification testing is meant to confirm that the verifier accepts all valid parse trees, and rejects all invalid ones according to the specifications of our language.

Data type Verification

Data type verification testing is meant to test the declaration, define value, relative computation, functions about our special data type. (res, bit)

Typical Test Example List (in the ec_test_case folder)

.ec file	.java file	Description
test_add.ec	test_add_result.java	Semantic Verification
test_bit.ec	test_bit_result.java	bit data type Verification
test_chip1.ec	test_chip1_result.java	bit data type Verification
test_delta_star.ec	test_delta_star_result.java	res data type Verification
test_equal_res.ec	test_equal_res.java	res data type Verification/ Semantic Verification
test_fun1.ec	test_fun1.java	Semantic Verification
test_fun2.ec	test_fun2.java	Semantic Verification
test_fun3.ec	test_fun3.java	Semantic Verification
test_gcd.ec	test_gcd_result.java	Semantic Verification
test_hello_world.ec	test_hello_world_result.java	Semantic Verification
test_res.ec	test_res_result.java	res data type Verification
test_show.ec	test_show.java	Semantic Verification

lesson Learned

Liming Zhang

- Learn how to write a complier using Ocaml. We can use ocamllex for lexical analysis and design our own language tokens; use ocamlyacc for syntax analysis and define the format of expression, statement, declaration and the whole program; use ocaml for semantically analysis and code generation and check the type and scope of the language.

- ✧ Learn Ocaml. Ocaml as a function language is very interesting and totally different from common language like C or Java. It mainly considers your code as a function and returns its value with a specific type. Thus, importance should be attached to your data type since Ocaml does not explicitly indicate data type when you declare a variable. And it usually uses List for iteration, pattern match for if-else situations. Moreover, its initial value is located in the end of "let..in..", which is quite different from object-oriented language.
- ✧ Teamwork. We really enjoy the atmosphere when working with the project together. We start from discussing about the idea of our own language, deciding and coding our project together. This is an enjoyable project.

Xinyue Zhou

- ✧ Participating in PLT project is both challenging and exciting, and I learned so much from this. Cooperating with team members requires patience, collaboration, and devotion. Beyond this is to maintain well managed version control so that so many version of our compiler created by different parts of our group would not be messed up. When implementing our project, our team met frequently to discussion the detail of our language and solve difficult OCaml problem together. I used to work until to the 4 a.m. for so many times just to solve one bug. Though the process is tough, but once we solved the problem or debug was successful in the late midnight, the happiness is inexpressible.

- ✧ In this final project, I learned and practiced how to implement a compiler

step by step from scratch for the language designed by us. It really seems to be impossible at the beginning to imagine we could finish such task that seems to be so difficult but finally we made it. I not only learned how to put the principle we learnt in the PLT class into practice, but also understand the rigorousness when designing a language in a deeper level. I really appreciated the professor and TAs' help and the support and help from my teammates and friends.

Lei Zhang

- How to thinking as a functional language
- If you don't know how tricky the work will be , then do it as eraly as possible.
- Team collaboration

Yingnan Li

- During this course, we not only learned a lot of stuff about how to construct the complier step by step. Also we get a deeper understanding about the base of those programming languages we usually use.
- From scanner parser Ast, throughout type checking process, we get Semantics AST and we use the translator to get the java source code. And in order to get what we really want, what we need is just to compile the java source code in the whole process, we know how to deal with a bunch of problems we have met but we have never imaged that.

- ✧ Ocaml is the most amazing language I have known in my entire life. When we write the Ocaml code, we r always between "OMG. Finally, it works out." and "r u kidding me?" we also learned that if u do not want to be totally lost in our own code, u should do a lot of work on notations, and comments.
- ✧ Last but not least. We learn how to contribute to our project. I know, there are a lot of pressures during the whole development of this project. But our guys are kind of enjoying it. We share everything. I also feel so happy to meet other four guys.

Wei Zhang

language design is very important, before writing the scanner, parser, and type checking we must keep in mind the specialty of our language.
need to put thought into practice, there are some unforeseen mistakes, which can only be discovered through actual testing.

Complete List

Scanner.mll

```
{ open Parser }

(* letter, digit *)
let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let quotes=['"']
```

```
let character = ['\b' '\t' '\n' '\r' '\\' ' ' '!' '@' '#' '$' '%' '^'
'&' '*' '(' ')' '-' '_' '+' '=' '{' '[' '}' ']' '|' ';' ':' '<' '>'
'.' ',' '?' '/' '~' '`']
          | letter | digit

rule token = parse
   [' ' '\t' '\r' '\n']  { token lexbuf }         (*Whitespace*)
   | "/*"     { comment lexbuf }          (* Comments *)
  | "//"     { line_comment lexbuf }
   (* keywords *)
   |"if"       { IF }
  |"else"      { ELSE }
  |"for"       { FOR }
  |"while"     { WHILE }
  |"return"    { RETURN }
  |"int"       { INT }
  |"float"     { FLOAT }
  |"string"    { STRING }
  |"new"       { NEW }
  |"switch"    { SWITCH }
  |"case"      { CASE }
  |"res"       { RES }
  |"bit"       { BIT }
  |"function"  { FUNCTION }
  |"default"   { DEFAULT  }
  |"value"     { VALUE }
  |"void"      { VOID }
  |"boolean"   { BOOLEAN }
  (* Operators *)
  |'+'        { PLUS }
```

```
|'-'          { MINUS }
|'*'          { MULTIPLY }
|'/'          { DIVIDE }
|'%'          { MODULUS }
|'~'          { RECIPROCAL }
|'<'          { LT }
|'>'          { GT }
|"<="         { LE }
|">="         { GE }
|"!="         { NE }
|"=="         { EQ }
|'!'          { NOT }
|'^'          { XOR }
|"!^"         { XNOR }
|'&'          { AND }
|"!&"         { NAND }
|'|'          { OR }
|"!|"         { NOR }
|'='          { ASSIGN }
|'#'          { SERIES }
|'$'          { PARALLEL }
 (*Punctuation*)
  | '(' { LPAREN }
| ')' { RPAREN }
  | '{' { LBRACE }
| '}' { RBRACE }
  | '[' { LBRACKET }
  | ']' { RBRACKET }
  | ';' { SEMICOLON }
| ',' { COMMA }
```

```
    | ':' { COLON }
    | '.' { DOT }

  | digit+  as integer    { INTLiteral(int_of_string integer) }
  | digit+ '.' digit*
  | '.' digit+ ('e' ['+''-']? digit+)?
  | digit+ ('.' digit+)? 'e' ['+' '-']? digit+   as float
{ FLOATLiteral(float_of_string float) }
    | quotes ['0' '1']+ quotes 'b' as var { BITLiteral (String.sub
var 0 (String.length var - 1) )  }
  | letter (letter | digit | '_')* as identifier { ID(identifier) }
| quotes (letter|digit|character)* quotes as sentence
{Sentence(sentence)}
    | eof { EOF }
  | _ as err_char { raise (Failure("illegal character " ^ Char.escaped
err_char)) }


(* comment *)
and line_comment = parse
    '\n'     { token lexbuf }
    | _      { line_comment lexbuf }

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

parser.mly

```
%{
  open Ast
  let parse_error s = Printf.ksprintf failwith "ERROR: %s" s
%}


%token LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET SEMICOLON COMMA
COLON DOT
%token IF ELSE FOR WHILE RETURN NEW SWITCH CASE FUNCTION DEFAULT VALUE
%token INT FLOAT STRING VOID BOOLEAN
%token LT GT LE GE NE EQ NOT XOR XNOR AND NAND OR NOR ASSIGN
%token PLUS MINUS MULTIPLY DIVIDE MODULUS
%token RECIPROCAL
%token RES BIT
%token SERIES PARALLEL
%token <int> INTLiteral
%token <float> FLOATLiteral
%token <string> ID
%token <string> BITLiteral
%token <string> Sentence

%token EOF


%nonassoc NOELSE
%nonassoc ELSE
```

```
%right ASSIGN
%left SERIES PARALLEL
%left OR NOR
%left XOR XNOR
%left AND NAND
%left EQ NE
%left GT GE LT LE
%left PLUS MINUS
%left MULTIPLY DIVIDE MODULUS
%right NOT RECIPROCAL
%nonassoc LBRACE
%left DOT

%start program
%type <Ast.program> program

%%

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

fdecl:
   type_decl ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list
RBRACE
     { { rtype   = $1;
         fname   = $2;
           formals = $4;
           locals  = List.rev $7;
```

```
          body    = $8
        }
      }

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

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

formal:
    type_decl ID
    { { vtype = $1;
        vname = $2;
    } }

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

vdecl:
    type_decl ID SEMICOLON
    { { vtype = $1;
        vname = $2
      }
    }
```

```
type_decl:
   INT     { Int }
  | FLOAT   { Float }
  | STRING  { String }
  | BIT     { Bit }
  | RES     { Res }
  | VOID    { Void }
  | BOOLEAN { Boolean }


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



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

```
expr:
     literal              { Literal($1) }

   | expr PLUS expr       { Binop($1, Add, $3) }
   | expr MINUS expr      { Binop($1, Sub, $3) }
   | expr MULTIPLY expr      { Binop($1, Mult, $3) }
   | expr DIVIDE expr     { Binop($1, Div, $3) }
    | expr GT expr        { Binop($1, Gt, $3) }
      |  expr NE expr        { Binop($1, Ne, $3) }
      |  expr LT expr        { Binop($1, Lt, $3) }
      |   expr LE expr        { Binop($1, Le, $3) }
      | expr GE expr        { Binop($1, Ge, $3) }
      | expr EQ expr        { Binop($1, Eq, $3) }

   | expr SERIES expr     { Binop($1, Series, $3)    }
   | expr PARALLEL expr   { Binop($1, Parallel, $3)  }
      | expr AND expr     { Binop($1, And, $3)    }
   | expr OR expr        { Binop($1, Or, $3)    }

   | RECIPROCAL expr     { Unary(Reciprocal, $2)}

   | PLUS expr           { Unary(Plus, $2) }
   | MINUS expr          { Unary(Minus, $2) }


   | lvalue                          { $1 }
   | lvalue ASSIGN expr                { Assign($1, $3) }

   | LPAREN expr RPAREN                { $2 }
```

```
    | ID LPAREN arg_list RPAREN          { Call($1, List.rev $3) }


lvalue:
    ID                                  { Id($1) }
   | expr DOT VALUE                          { Afterop($1, Dot)}


literal:
    INTLiteral                                  { IntLit($1) }
    | FLOATLiteral                               { FloatLit($1) }
    | BITLiteral                                { BinaryLit($1) }
    | Sentence                                 {SentenceLit($1)}

arg_list:
     /* nothing */                      { [] }
    | expr                             { [$1] }
    | arg_list COMMA expr                  { $3::$1 }
```

ast.ml

```
type op = Add | Sub | Mult | Div | Series | Parallel |Gt |Ne |Lt |Le
|Ge | Eq | And | Or
type uop = Reciprocal | Plus | Minus

type afterop = Dot

type datatype = Int | Float | String | Bit | Res | Void | Boolean

type literal =
    IntLit of int                         (* 42 *)
  | FloatLit of float                     (* 3.4 *)
  | BinaryLit of string                   (* 010101b *)
  | SentenceLit of string                 (*"Hello World!"*)

type expr =                               (* Expressions *)
    Literal of literal
  | Id of string                          (* foo *)
  | Binop of expr * op * expr             (* a + b *)
  | Unary of uop * expr                   (* !b *)
  | Afterop of expr * afterop             (* r1.value *)
  | Assign of expr * expr                 (* a = b *)
  | Call of string * (expr list)          (* foo(1, 25) *)
  | Noexpr                                (* While() *)

type var_decl = {
    vtype: datatype;
    vname: string;
}
```

```
type stmt =                                 (* Statements  nothing *)
    Block of (var_decl list) * (stmt list)
  | Expr of expr                            (* foo = bar + 3; *)
  | Return of expr                          (* return 42 *)
  | If of expr * stmt * stmt                 (* if (foo == 42) {} else {}
*)
  | For of expr * expr * expr * stmt         (* for loop *)
  | While of expr * stmt                    (* while (i<10) { i = i + 1 }
*)



type func_decl = {
    rtype : datatype;
    fname : string;
    formals : var_decl list;
    locals : var_decl list;
    body : stmt list;
  }

type program = var_decl list  * func_decl list
```

sast.ml

```
open Ast


type var_decl = {
   vvtype: datatype;
   vvname: string;
}

type simple_expr =                                (* Expressions *)
   Literal of literal
  | Id of var_decl                            (* foo *)
  | Binop of expr * op * expr                 (* a + b *)
  | Unary of uop * expr                       (* !b *)
  | Afterop of expr * afterop                 (* r1.value *)
  | Assign of expr * expr                     (* a = b *)
  | Call of string * (expr list)              (* foo(1, 25) *)
  | Noexpr                                  (* While() *)
and expr = simple_expr * datatype

type stmt =                                 (* Statements  nothing *)
   Block of (var_decl list) * (stmt list)
  | Expr of expr                            (* foo = bar + 3; *)
  | Return of expr                          (* return 42 *)
  | If of expr * stmt * stmt                 (* if (foo == 42) {} else {}
*)
  | For of expr * expr * expr * stmt        (* for loop *)
  | While of expr * stmt                    (* while (i<10) { i = i + 1 }
*)
```

```
type func_decl = {
    frtype : datatype;
    ffname : string;
    fformals : var_decl list;
    flocals : var_decl list;
    fbody : stmt list;
  }

type program = var_decl list  * func_decl list
```

typecheck.ml

```
open Ast
open Sast


let string_of_op = function
      Add -> "+"
    | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Series -> "#"
  | Parallel -> "$"
  | Eq -> "=="
  | Ne -> "!="
  | Gt -> ">"
  | Lt -> "<"
  | Ge -> ">="
  | Le -> "<="

let rec string_of_obj_type t = match t with
    Int -> "int"
  | Float -> "float"
  | String -> "string"
  | Bit -> "bit"
  | Res -> "res"
  | Void -> "void"
  | Boolean -> "boolean"

type symbol_table = {
```

```
  parent : symbol_table option;
  variables : Sast.var_decl list;
  functions: Sast.func_decl list;
}

type trans_env = {
  scope : symbol_table;
}

let rec find_variable (scope : symbol_table) name =
  try
    List.find (fun v -> v.vvname = name) scope.variables
  with Not_found ->
    match scope.parent with
    Some(parent) -> find_variable parent name
    | _ -> raise (Failure("variable " ^ name ^ " not defined"))

let var_exists scope name =
    try
      let _ = find_variable scope name
      in true
    with Failure(_) ->
      false

let rec find_function (scope : symbol_table) name = match name with
    "show" ->  {frtype = Void; ffname =
"show";fformals=[];flocals=[];fbody=[] }
  | _ -> (try
    List.find (fun f -> f.ffname = name) scope.functions
  with Not_found ->
```

```
    match scope.parent with
    Some(parent) -> find_function parent name
    | _ -> raise (Failure("function " ^ name ^ " not defined")))


let func_exists scope name =
    List.exists (fun f -> f.ffname = name) scope.functions

let assign_allowed lt rt = match lt with
      Res ->  (rt = Int) || (rt = Float) || (rt = Res)
    | Bit ->  (rt = Bit) || (rt = String)
    | _ -> lt = rt

let rec can_assign lt rval =
  let (_, rt) = rval in
    if assign_allowed lt rt then
      rval
    else
      raise (Failure("type " ^ string_of_obj_type rt ^ " cannot be put
into type " ^ string_of_obj_type lt))

let to_ast ast_vdecl sast_vdecl =
  {vtype = sast_vdecl.vvtype; vname = sast_vdecl.vvname}


let can_op lval op rval =
  let (_, lt) = lval
  and (_, rt) = rval in
  let type_match = (lt = rt) in
  let int_match = (lt = Int) in
```

```
  let float_match = (lt = Float) in
  let string_match = (lt = String) in
  let bit_match = (lt = Bit) in
  let res_match = (lt = Res) in
  let boolean_match = (lt = Boolean) in
  let result = match op with
      Ast.Add    -> (type_match && (int_match || float_match ||
string_match || bit_match || res_match )), lt
    | Ast.Sub    -> (type_match && (int_match || float_match ||
bit_match || res_match ) ), lt
    | Ast.Mult   -> (type_match && (int_match || float_match ||
bit_match || res_match ) ), lt
    | Ast.Div    -> (type_match && (int_match || float_match ||
bit_match || res_match ) ), lt
    | Ast.Series  -> (type_match && res_match ), lt
    | Ast.Parallel  -> (type_match && res_match), lt
    | Ast.Eq -> (type_match && (int_match || float_match || bit_match
|| res_match )), Boolean
    | Ast.Ne -> (type_match && (int_match || float_match || bit_match
|| res_match)), Boolean
    | Ast.Gt -> (type_match && (int_match || float_match || bit_match
|| res_match)), Boolean
    | Ast.Lt -> (type_match && (int_match || float_match || bit_match
|| res_match)), Boolean
    | Ast.Ge -> (type_match && (int_match || float_match || bit_match
|| res_match)), Boolean
    | Ast.Le -> (type_match && (int_match || float_match || bit_match
|| res_match)), Boolean

  in if fst result then
```

```
    snd result
  else
    raise (Failure("operator" ^ string_of_op op ^ " cannot be used on
types " ^
      string_of_obj_type lt ^ " and " ^ string_of_obj_type rt))

let translate (globals, funcs) =
  let rec trans_expr env = function
      Ast.Id(n) -> let vdecl = (find_variable env.scope n) in
                   Sast.Id(vdecl), vdecl.vvtype
     | Ast.Unary (un_op, e) ->
                    let et = trans_expr env e in
                      let _,t = et in            (*expr and type*)
                        let tt = match un_op with
                          | Reciprocal -> if t = Int or t = Float
then t else
                              (*TODO: t!=0*)
                                raise (Failure ("Only integers
and floats are allowed for using reciprocal"))
                        (*  | Not -> if t = Boolean then Boolean else
                                raise (Failure ("Only boolean is
allowed for boolean operators"))  *)
                     | Plus -> if t = Int or t = Float then t else
                                raise (Failure ("Only integers
and floats can be added a positive sign"))
                     | Minus -> if t = Int or t = Float then t else
                                raise (Failure ("Only integers
and floats can be added a negative sign"))
                     | _ -> raise (Failure ("The operator is not
unary"))
```

```
                in
                Sast.Unary(un_op, et), tt
          | Ast.Afterop (e, after_op) ->
               let et = trans_expr env e in
               let _,t = et in
                   let tt = match after_op with
                      | Dot -> if t = Res or t = Bit then t else
                          (*TODO: t!=0*)
                             raise (Failure ("Only res or bit is
allowed for dot value."))
                      | _ -> raise (Failure ("The operator is not
afterop."))
                in
                Sast.Afterop (et, after_op), tt
         | Ast.Literal(lit) -> (match lit with
                 | IntLit i -> Literal(lit), Int
                 | FloatLit f -> Literal(lit), Float
                 | BinaryLit b -> Literal(lit), Bit
                 | SentenceLit s -> Literal(lit), String
          )
         | Ast.Binop(e1, op, e2) ->
             let e1 = trans_expr env e1
             and e2 = trans_expr env e2
             in let rtype = can_op e1 op e2 in
             Sast.Binop(e1, op, e2), rtype

        | Ast.Call (func_name, params) ->
            let func_decl_call =
            (find_function env.scope func_name)
            in
```

```
        let checked_fname = if (func_decl_call.ffname=func_name)
then func_name
           else raise (Failure("undeclaration of " ^ func_name))
        in
        let typed_params = List.map (trans_expr env) params
        in
            Sast.Call(checked_fname, typed_params),
func_decl_call.frtype

      | Ast.Assign(lv, e) ->
         let lval, t = (trans_expr env lv) in
         let aval = (trans_expr env e) in
         Sast.Assign((lval, t), (can_assign t aval)), t
      | Ast.Noexpr ->
         Sast.Noexpr, Void

    in let add_local env v =
         let evalue = if (var_exists env.scope v.vname) = true then
            raise (Failure("redeclaration of " ^ v.vname))
         in (**)
         let new_v = {
               vvname = v.vname;
               vvtype = v.vtype;

           }
         in let vars = new_v :: env.scope.variables
         in let scope' = {env.scope with variables = vars}
         in {(*env with*) scope = scope'}

   in let rec trans_stmt env = function
```

```
      Ast.Block(v, s) ->
        let scope' = {parent = Some(env.scope); variables = [];
functions = []}
          in let env' = {(*env with*) scope = scope'}
          in let block_env = List.fold_left add_local env' (List.rev
v)

          in let s' = List.map (fun s -> trans_stmt block_env s) s
          in let fvlist = block_env.scope.variables
          in Sast.Block(fvlist, s')

      | Ast.Expr(e) ->
          Sast.Expr(trans_expr env e)
      | Ast.Return(e) ->
          Sast.Return(trans_expr env e)
      | Ast.If (e, s1, s2) ->
          let e' = trans_expr env e
          in Sast.If(can_assign Boolean e', trans_stmt env s1,
trans_stmt env s2)
      | Ast.While (e, s) ->
          let e' = trans_expr env e
          in Sast.While(can_assign Boolean e', trans_stmt env s)
      | Ast.For (e1, e2, e3, s) ->
          let e2' = trans_expr env e2
          in Sast.For(trans_expr env e1, can_assign Boolean e2',
trans_expr env e3, trans_stmt env s)

  in let add_func env f =
```

```
      let new_f = match ((var_exists env.scope f.fname) ||
(func_exists env.scope f.fname)) with
        true -> raise (Failure("redeclaration of " ^ f.fname))
      | false ->
          let scope' = {parent = Some(env.scope); variables = [];
functions = []}
          in let env' = {(*env with*) scope = scope'}
          in let env' = List.fold_left add_local env' (List.rev
f.formals)
          in {
            frtype = f.rtype;
            ffname = f.fname;
            fformals = env'.scope.variables;
            flocals = [];
            fbody = [];
          }
      in let funcs = new_f :: env.scope.functions
      in let scope' = {env.scope with functions = funcs}
      in {(*env with*) scope = scope'}

    in let trans_func env (f : Ast.func_decl) =
      let sf = find_function env.scope (f.fname)
     in let functions' = List.filter (fun f -> f.ffname != sf.ffname)
env.scope.functions
      in let scope' = {parent = Some(env.scope); variables =
sf.fformals; functions = []}
      in let env' = {(*env with*) scope = scope'}
      in let formals' = env'.scope.variables
      in let env' = List.fold_left add_local env' (f.locals)
      in let remove v =
```

```
          not (List.exists (fun fv -> fv.vvname = v.vvname) formals')
        in let locals' = List.filter remove env'.scope.variables
        in let body' = List.map (fun f -> trans_stmt env' f) (f.body)
        in let new_f = {
          sf with
          fformals = formals';
          flocals = locals';
          fbody = body';
        }
        in let funcs = new_f :: functions'
        in let scope' = {env.scope with functions = funcs}
        in {(*env with*) scope = scope'}

      in let validate_func f =
        let is_return = function
            Sast.Return(e) -> true
          | _ -> false
        in let valid_return = function
            Sast.Return(e) -> if assign_allowed f.frtype (snd e) then
                                true
                              else
                                raise (Failure(  f.ffname ^ " must return
type " ^
                                 string_of_obj_type f.frtype ^
                                 ", not " ^ string_of_obj_type (snd e)
                                ))
          | _ -> false
        in let returns = List.filter is_return f.fbody
        in let _ = List.for_all valid_return returns
        in let return_count = List.length returns
```

```
      in if (return_count = 0 && f.frtype != Void) then
        raise (Failure(f.ffname ^ " must have a return type of " ^
string_of_obj_type f.frtype))
      else if List.length f.fformals > 8 then
        raise (Failure(f.ffname ^ " must have less than 8 formals"))
      else
        f

  in let global_scope = {
          parent = None;
          variables = [];
          functions = [];
        }
    in let genv = {
          scope = global_scope;
        }
        in let genv = List.fold_left add_local genv (List.rev
globals)
        in let genv = List.fold_left add_func genv (List.rev funcs)
        in let genv = List.fold_left trans_func genv (List.rev funcs)
        in if func_exists genv.scope "main" then
          (genv.scope.variables, List.map validate_func
genv.scope.functions)
        else
          raise (Failure("no main function defined"))
```

translator.ml

```
open Sast


let package_del = "package ec;" (* Package declaration. *)
let import_decl = "" (* Import needed packages. Not needed now. *)
(* Java code of the defination of the inner class of Res. *)
let res_def= "\n public static class Res {\n double value; \n public
Res(){ \n  super(); \n  } \n public Res(double name){ \n
this.value=name; \n } \n }"
(* Java code of the defination of the inner class of Bit. *)
let bit_def="  public static class Bit {
      String value;
      public static boolean[] BinstrToBool(String input) {
         boolean[] output = new boolean[input.length()];
         for (int i = 0; i < input.length(); i++)
            if (input.charAt(i) == '1')
               output[i] = true;
            else if (input.charAt(i) == '0')
               output[i] = false;
         return output;
      }
   }
"
(* Java code of getParallel method which compute parallel resistance.
*)
let getParallel_fun_def="\n public static Res getParallel (Res r1,Res
r2){ \n Res rp=new Res(); \n rp.value=1/(1/r1.value+1/r2.value); \n
return rp; \n }"
```

```
(* Java code of getSerial method which compute serially connected
resistance. *)
let getSerial_fun_def="\n public static Res getSerial (Res r1,Res
r2){ \n Res rs=new Res(); \n rs.value=r1.value+r2.value; \n
return rs; \n    }"
(* Java code of method that computes AND operation for two BIT type
variables method. *)
let  compute_AND_fun_def= "public static Bit aANDb(Bit a, Bit b) {
        Bit resultBit = new Bit();
        int lmin = Math.min(a.value.length(), b.value.length());
        int lmax = Math.max(a.value.length(), b.value.length());
        boolean[] result = new boolean[lmin];
        for (int i = 0; i < lmin; i++) {
            result[i] = Bit.BinstrToBool(a.value)[i]
                    && Bit.BinstrToBool(b.value)[i];
        }

        String resultStr = new String();
        for (int i = 0; i < lmin; i++) {
            if (result[i])
                resultStr = resultStr + \"1\";
            else
                resultStr = resultStr + \"0\";
        }
        for (int i = 0; i < lmax - lmin; i++) {
            resultStr = resultStr + \"0\";
        }
        resultBit.value = resultStr;
        return resultBit;
    }"
```

```
(* Java code of method that computes OR operation for two BIT type
variables method. *)

let compute_OR_fun_def="public static Bit aORb(Bit a, Bit b) {
      Bit resultBit = new Bit();
      int lmin = Math.min(a.value.length(), b.value.length());
      int lmax = Math.max(a.value.length(), b.value.length());
      boolean[] result = new boolean[lmin];
      for (int i = 0; i < lmin; i++) {
         result[i] = Bit.BinstrToBool(a.value)[i]
               || Bit.BinstrToBool(b.value)[i];
      }

      String resultStr = new String();
      for (int i = 0; i < lmin; i++) {
         if (result[i])
            resultStr = resultStr + \"1\";
         else
            resultStr = resultStr + \"0\";
      }
      for (int i = 0; i < lmax - lmin; i++) {
         resultStr = resultStr + \"0\";
      }
      resultBit.value = resultStr;
      return resultBit;
   }"
(* Java code of main method declaration *)
let main_fdecl = "\n public static void main(String[] args) throws
Exception\n{\nECStart();\n}\n"
```

```
(* Concatenate the java code of pre-defined inner class Res and
functions *)
let begin_code_necessary= main_fdecl
let begin_code_with_res= res_def  ^ getParallel_fun_def ^
getSerial_fun_def   ^ main_fdecl
let begin_code_with_bit=  bit_def ^   compute_AND_fun_def ^
compute_OR_fun_def ^ main_fdecl
let begin_code_with_res_and_bit= res_def ^ bit_def ^
getParallel_fun_def ^ getSerial_fun_def ^ compute_AND_fun_def ^
compute_OR_fun_def ^ main_fdecl

(*27*)let jstring_of_datatype dtype =
  match dtype with
    Ast.Int -> "int"
  |  Ast.Float -> "double"
  |  Ast.Boolean -> "boolean"
  |  Ast.Void -> "void"
  | Ast.String -> "String"
    | Ast.Res -> "Res"
    |  Ast.Bit ->"Bit"          (*å¾⬆å➡↓astå¯¹å°⬇*)

(*37*)let rec jstring_of_expr global_vars local_vars = function
    Literal(ll)->
        (match ll with
         Ast.IntLit(i)-> string_of_int i
        | Ast.FloatLit(f)-> string_of_float f
        |  Ast.BinaryLit(b)-> b
        |  Ast.SentenceLit(s)-> s)
    |Id(l)->l.vvname
    | Binop(et1, o, et2) ->
```

```
      let (e1,eu1) =et1 in
      let (e2,eu2) =et2 in
       (match o with
      |   Ast.Add -> jstring_of_expr global_vars local_vars e1 ^ "+"
^ " " ^ jstring_of_expr global_vars local_vars e2
      |   Ast.Sub -> jstring_of_expr global_vars local_vars e1 ^ "-"
^ " " ^ jstring_of_expr global_vars local_vars e2
      |   Ast.Mult -> jstring_of_expr global_vars local_vars e1 ^ "*"
^ " " ^ jstring_of_expr global_vars local_vars e2
      |   Ast.Div -> jstring_of_expr global_vars local_vars e1 ^ "/"
^ " " ^ jstring_of_expr global_vars local_vars e2
      |   Ast.Gt ->jstring_of_expr global_vars local_vars e1 ^ ">"
^ " " ^ jstring_of_expr global_vars local_vars e2
      |   Ast.Ne ->jstring_of_expr global_vars local_vars e1 ^ "!="
^ " " ^ jstring_of_expr global_vars local_vars e2
      |   Ast.Lt ->jstring_of_expr global_vars local_vars e1 ^ "<"
^ " " ^ jstring_of_expr global_vars local_vars e2
      |   Ast.Le ->jstring_of_expr global_vars local_vars e1 ^ "<="
^ " " ^ jstring_of_expr global_vars local_vars e2
      |   Ast.Ge ->jstring_of_expr global_vars local_vars e1 ^ ">="
^ " " ^ jstring_of_expr global_vars local_vars e2
      |   Ast.Eq ->jstring_of_expr global_vars local_vars e1 ^ "=="
^ " " ^ jstring_of_expr global_vars local_vars e2
      |   Ast.Series -> " getSerial(" ^ jstring_of_expr global_vars
local_vars  e1 ^ ","  ^ " " ^ jstring_of_expr global_vars local_vars
e2 ^ ")"
      |   Ast.Parallel -> " getParallel(" ^ jstring_of_expr
global_vars local_vars  e1 ^ ","  ^ " " ^ jstring_of_expr global_vars
local_vars e2 ^ ")"
```

```
      |   Ast.And -> " aANDb(" ^ jstring_of_expr global_vars
local_vars  e1 ^ ","  ^ " " ^ jstring_of_expr global_vars local_vars
e2 ^ ")"
      |   Ast.Or ->"aORb(" ^ jstring_of_expr global_vars local_vars
e1 ^ ","  ^ " " ^ jstring_of_expr global_vars local_vars e2 ^ ")"
      )
   |Unary (op,et)->
      let (e,eu)=et in
      (match op with
      | Ast.Reciprocal -> "(1/" ^ jstring_of_expr global_vars
local_vars e ^")"
      |  Ast.Plus->  "(+" ^ jstring_of_expr global_vars local_vars
e ^")"
      |  Ast.Minus->  "(-" ^ jstring_of_expr global_vars local_vars
e ^")")
   | Afterop (ent,aft)-> let (en,eu)=ent in jstring_of_expr
global_vars local_vars en ^".value "
   |  Assign (at,bt) -> let (a,au)=at in let (b,bu)=bt in
jstring_of_expr global_vars local_vars a ^"="^ jstring_of_expr
global_vars local_vars b
 | Call(f, elt) ->
     (match f with
      "show" -> "System.out.println(" ^ (String.concat "" (List.map
(jstring_of_expr global_vars local_vars) (List.map (  fun
(a1,a2)->a1) elt) )) ^ ")"

    | _ -> f ^ "(" ^ String.concat ", " (List.map (jstring_of_expr
global_vars local_vars) (List.map (   fun (a1,a2)->a1) elt)) ^ ")"
     )
```

```
  | Noexpr -> ""

let rec jstring_of_stmt global_vars local_vars = function
    Block(decls, stmts) ->
      "{\n" ^ String.concat "" (List.map (jstring_of_stmt global_vars
local_vars) stmts) ^ "}\n"
  | Expr(exprt) -> let (expr,expru)=exprt in jstring_of_expr
global_vars local_vars expr ^ ";\n";
  | Return(exprt) -> let (expr,expru)=exprt in  "return " ^
jstring_of_expr global_vars local_vars expr ^ ";\n";
  | If(et, s, Block([],[])) ->  let (e,eu)=et in "if (" ^
jstring_of_expr global_vars local_vars e ^ ")\n" ^ jstring_of_stmt
global_vars local_vars s
  | If(et, s1, s2) -> let (e,eu)=et in "if (" ^ jstring_of_expr
global_vars local_vars e ^ ")\n" ^
      jstring_of_stmt global_vars local_vars s1 ^ "else\n" ^
jstring_of_stmt global_vars local_vars s2
  | For(e1t, e2t, e3t, s) -> let (e1,e1u)=e1t in let (e2,e2u)=e2t
in let (e3,e3u)=e3t in
      "for (" ^ jstring_of_expr global_vars local_vars e1  ^ " ; " ^
jstring_of_expr global_vars local_vars e2 ^ " ; " ^
      jstring_of_expr global_vars local_vars e3  ^ ") " ^
jstring_of_stmt global_vars local_vars s
  | While(et, s) ->  let (e,eu)=et in "while (" ^ jstring_of_expr
global_vars local_vars e ^ ") " ^ jstring_of_stmt global_vars
local_vars s


let jstring_of_vdecl vdecl = match (vdecl.vvtype) with
```

```
| Ast.Res -> (jstring_of_datatype vdecl.vvtype) ^ " " ^ vdecl.vvname
^ "=new "^ (jstring_of_datatype vdecl.vvtype) ^"()"^" ;\n"
| Ast.Bit ->  (jstring_of_datatype vdecl.vvtype) ^ " " ^
vdecl.vvname ^ "=new "^ (jstring_of_datatype vdecl.vvtype)
^"()"^" ;\n"
| _ -> (jstring_of_datatype vdecl.vvtype) ^ " " ^ vdecl.vvname ^ " ;\n"

let jstring_of_gvdecl gvdecl =
  "public static " ^ jstring_of_vdecl gvdecl

let jstring_of_formal formal =
  jstring_of_datatype formal.vvtype ^ " " ^ formal.vvname

let jstring_of_fdecl global_vars fdecl =
  let local_vars = (List.map (fun a -> { vvname = a.vvname; vvtype
= a.vvtype }) fdecl.fformals)
               @ (List.map (fun a -> { vvname = a.vvname; vvtype =
a.vvtype }) fdecl.flocals)
  in
  (match fdecl.ffname with
    "main" -> "static " ^ jstring_of_datatype fdecl.frtype  ^ "
ECStart()"
    | _ -> "static " ^ jstring_of_datatype fdecl.frtype ^ " " ^
fdecl.ffname ^
         "(" ^  String.concat ", " (List.map jstring_of_formal
fdecl.fformals) ^ ")"
   ) ^
  "\n{\n" ^
  String.concat "" (List.map jstring_of_vdecl fdecl.flocals) ^
```

```
  String.concat "" (List.rev(List.map (jstring_of_stmt global_vars
local_vars) fdecl.fbody)) ^
  "}\n"

let jstring_of_header (g_var, l_var)=

  let global_vars = List.map (fun a -> { vvname = a.vvname; vvtype
= a.vvtype }) g_var
    in
    let global_vars_type_list = List.map (fun a ->  a.vvtype )
global_vars
    in
    let funcs_locals_list=List.map ( fun a -> {frtype=a.frtype;
ffname=a.ffname;fformals=a.fformals;
flocals=a.flocals;fbody=a.fbody})l_var
    in
let funcs_locals_type_list_temp1 =List.map (fun a ->  a.flocals)
funcs_locals_list

in
let funcs_locals_type_list_temp2=List.concat
funcs_locals_type_list_temp1

in
    let funcs_locals_type_list =List.map(fun a->
a.vvtype)funcs_locals_type_list_temp2 in
```

```
   let funcs_formals_list=List.map ( fun a -> {frtype=a.frtype;
ffname=a.ffname;fformals=a.fformals;
flocals=a.flocals;fbody=a.fbody})l_var
   in
let funcs_formals_type_list_temp1 =List.map (fun a ->  a.fformals )
funcs_formals_list

in
let funcs_formals_type_list_temp2=List.concat
funcs_formals_type_list_temp1

in
   let funcs_formals_type_list =List.map(fun a->
a.vvtype)funcs_formals_type_list_temp2 in

let vars_type_list = global_vars_type_list
@(funcs_locals_type_list)@funcs_formals_type_list
  in
   if( List.mem  Ast.Res vars_type_list && List.mem  Ast.Bit
vars_type_list) then begin_code_with_res_and_bit
   else if( List.mem  Ast.Res vars_type_list) then
begin_code_with_res
   else if( List.mem  Ast.Bit vars_type_list) then
begin_code_with_bit
   else begin_code_necessary

let jstring_of_program (vars, funcs)  file_name  =
   let header=jstring_of_header (vars, funcs)in
  let global_vars = List.map (fun a -> { vvname = a.vvname; vvtype
= a.vvtype }) vars
```

```
  in
  package_del ^ "\n" ^ import_decl ^ "\n\n" ^
  "public class " ^ (String.sub file_name 0 ((String.length file_name)
- 3)) ^"_result"^
  "\n{\n" ^header  ^
  String.concat "" (List.map jstring_of_gvdecl vars) ^ "\n\n" ^
  String.concat "\n" (List.map (jstring_of_fdecl global_vars) funcs)
^
  "\n}"
```

ec.ml

```
type action = Sast | Compile | Translator | Typecheck | All


let _ =
  let action = if Array.length Sys.argv > 1 then
     List.assoc Sys.argv.(1) [ ("-a", Sast);
                                ("-j", Translator);
                                ("-t", Typecheck);
                                ("-c", All)]
  else Compile in
  let input_file = ref Sys.argv.(2) in
  let input = open_in !input_file in
  let lexbuf = Lexing.from_channel input in
  let program = Parser.program Scanner.token lexbuf in
  match action with

   Translator ->
    let listing =
     Translator.jstring_of_program (Typecheck_nobug.translate
program) !input_file
    in print_endline listing
  | All ->
     ignore (Typecheck_nobug.translate program);
     let listing =
          Translator.jstring_of_program (Typecheck_nobug.translate
program) !input_file
     in print_endline listing
```