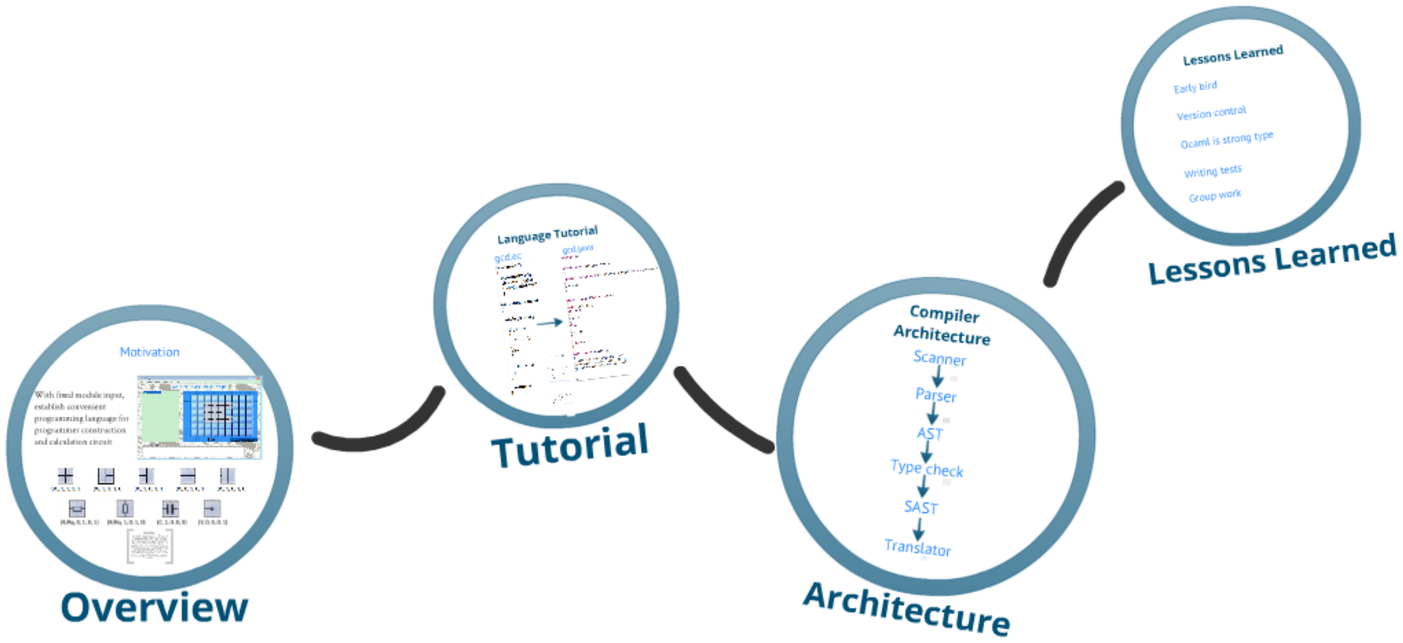
Easy Circuit

-An easy language to draw your circuit and figure it out



Motivation

With fixed module input, establish convenient programming language for programmer construction and calculation circuit

Overview

Language Tutorial

gcd.ec

gcd.java

Tutorial

Compiler Architecture

Scanner → Parser → AST → Type check → SAST → Translator

Architecture

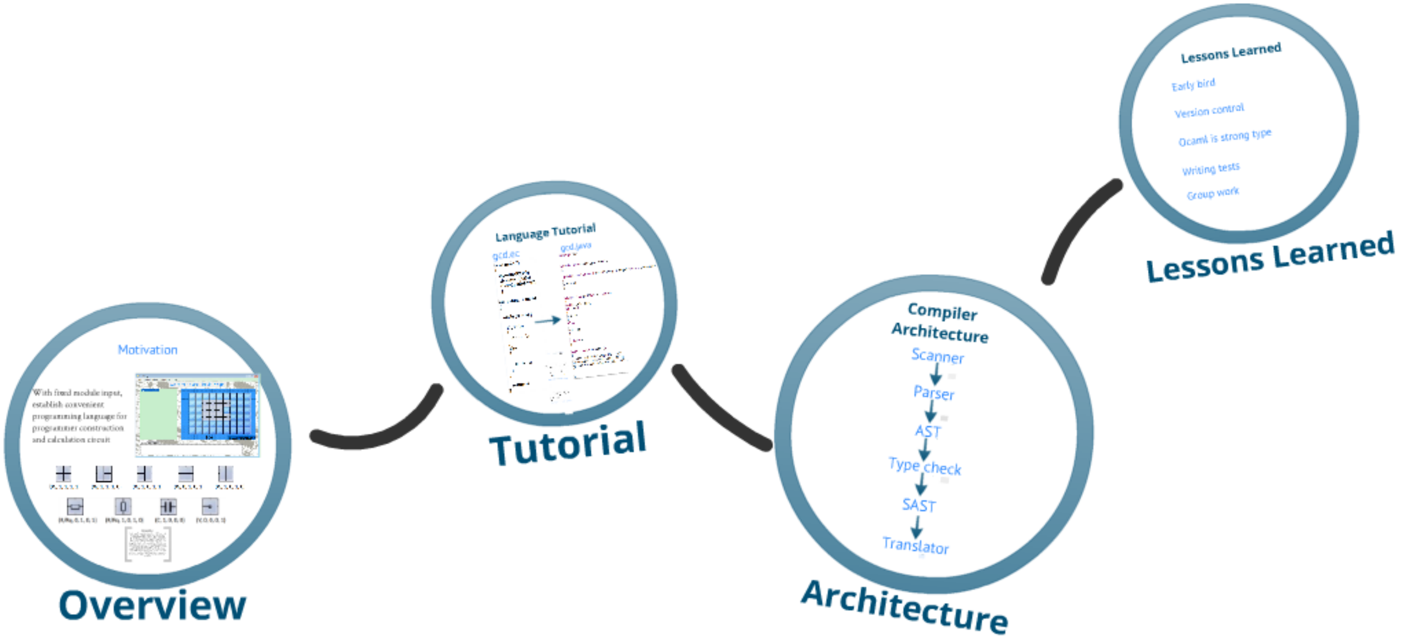
Lessons Learned

- Early bird
- Version control
- Ocaml is strong type
- Writing tests
- Group work

Lessons Learned

Easy Circuit

-An easy language to draw your circuit and figure it out



Motivation

With fixed module input, establish convenient programming language for programmer construction and calculation circuit

Overview

Language Tutorial

gcd.ec

gcd.java

Tutorial

Compiler Architecture

Scanner → Parser → AST → Type check → SAST → Translator

Architecture

Lessons Learned

Early bird

Version control

Ocaml is strong type

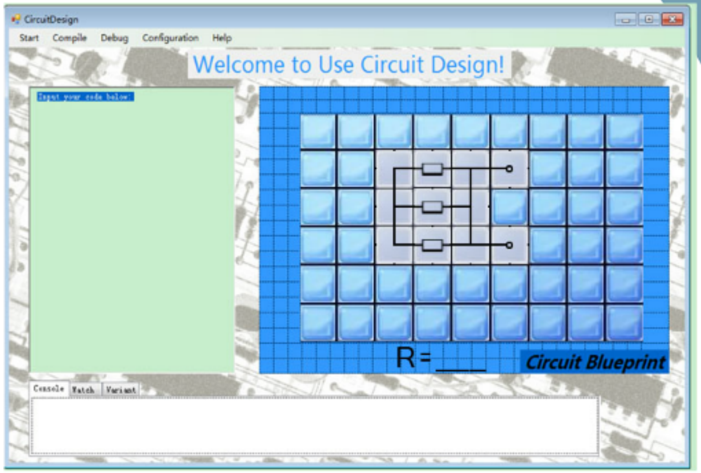
Writing tests

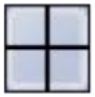
Group work

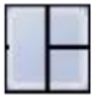
Lessons Learned

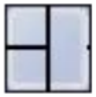
Motivation

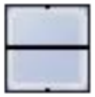
With fixed module input, establish convenient programming language for programmer construction and calculation circuit

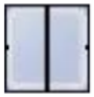


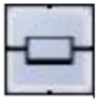

(W, 1, 1, 1, 1)

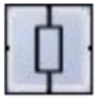

(W, 1, 1, 1, 0)

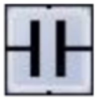

(W, 1, 0, 1, 1)

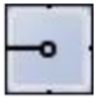

(W, 0, 1, 0, 1)


(W, 1, 0, 1, 0)


(R/Rq, 0, 1, 0, 1)


(R/Rq, 1, 0, 1, 0)


(C, 1, 0, 0, 0)


(V, 0, 0, 0, 1)

Introduction

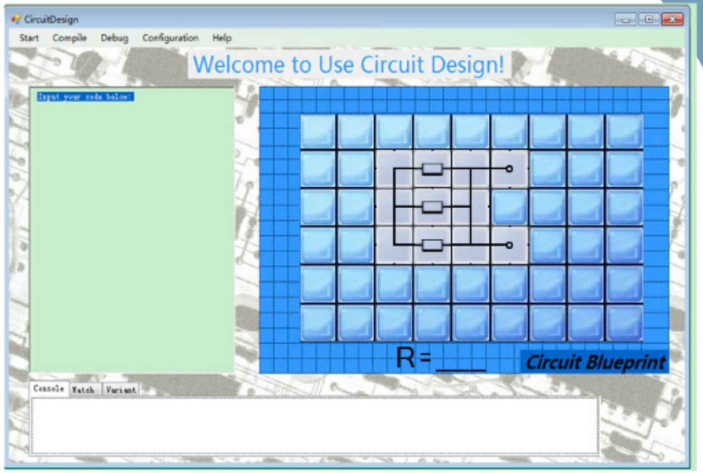
Circuit design plays an important role in many areas. However, some circuit operations, such as computing equivalent resistance, or transforming a circuit, though intuitionally in some ways, are still hard to formularly express in mathematics or programs. Famous electronic design automation (EDA) and simulation programs, such as Multisim, 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.

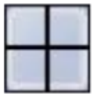
Introduction

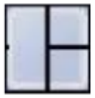
Circuit design plays an important role in many areas. However, some circuit operations, such as computing equivalent resistance, or transforming a circuit, though intuitionally in some ways, are still hard to formularily express in mathematics or programs. Famous electronic design automation (EDA) and simulation program, such as Multisim1, 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.

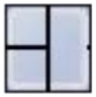
Motivation

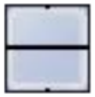
With fixed module input, establish convenient programming language for programmer construction and calculation circuit

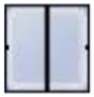


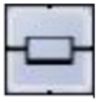

(W, 1, 1, 1, 1)

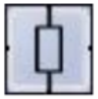

(W, 1, 1, 1, 0)

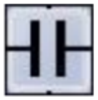

(W, 1, 0, 1, 1)

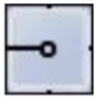

(W, 0, 1, 0, 1)


(W, 1, 0, 1, 0)


(R/Rq, 0, 1, 0, 1)


(R/Rq, 1, 0, 1, 0)


(C, 1, 0, 0, 0)

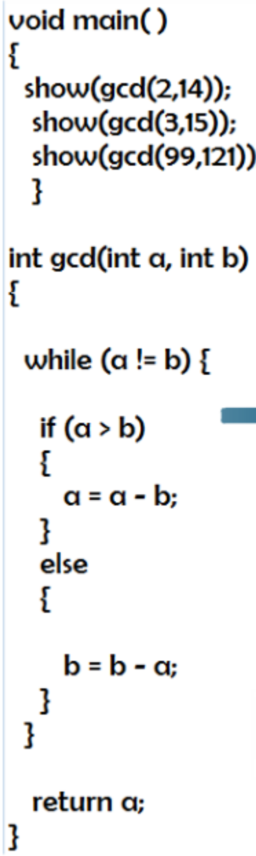

(V, 0, 0, 0, 1)

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 formularly express in mathematics or programs. Famous electronic design automation (EDA) and simulation programs, such as Multisim, 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 who 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.

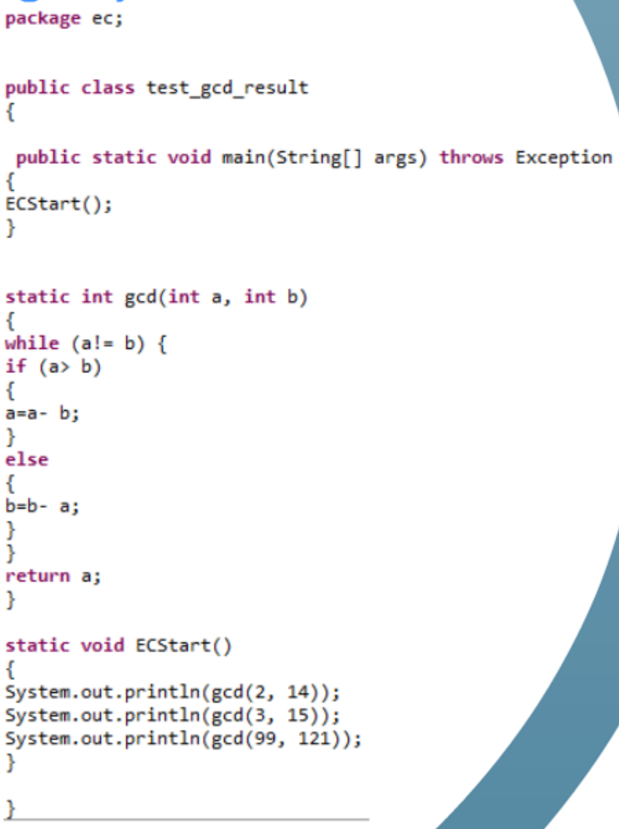
Language Tutorial

gcd.ec



```
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;
}
```

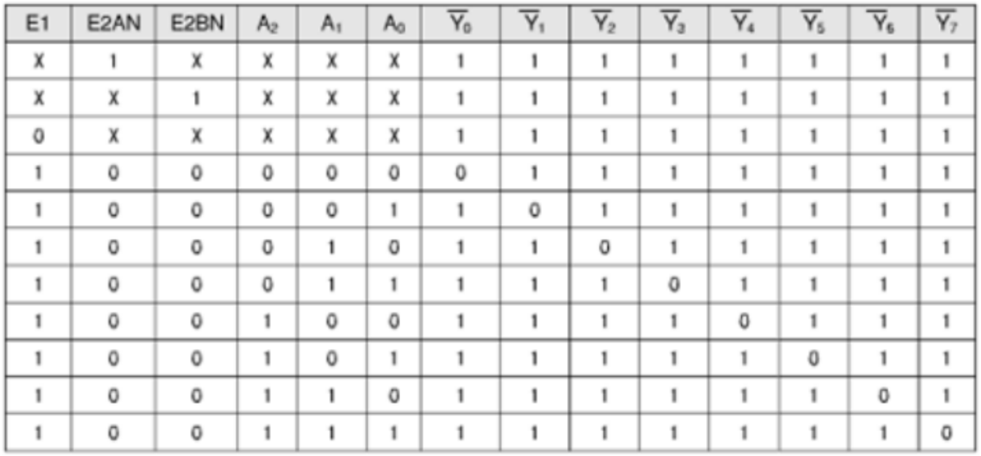
gcd.java



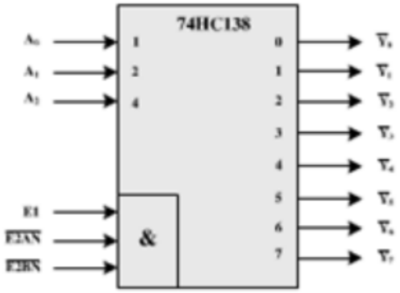
```
package ec;


public class test_gcd_result
{

 public static void main(String[] args) throws Exception
{
ECStart();
}


static int gcd(int a, int b)
{
while (a!= b) {
if (a> b)
{
a=a- b;
}
else
{
b=b- a;
}
}
return a;
}

static void ECStart()
{
System.out.println(gcd(2, 14));
System.out.println(gcd(3, 15));
System.out.println(gcd(99, 121));
}

}
```

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

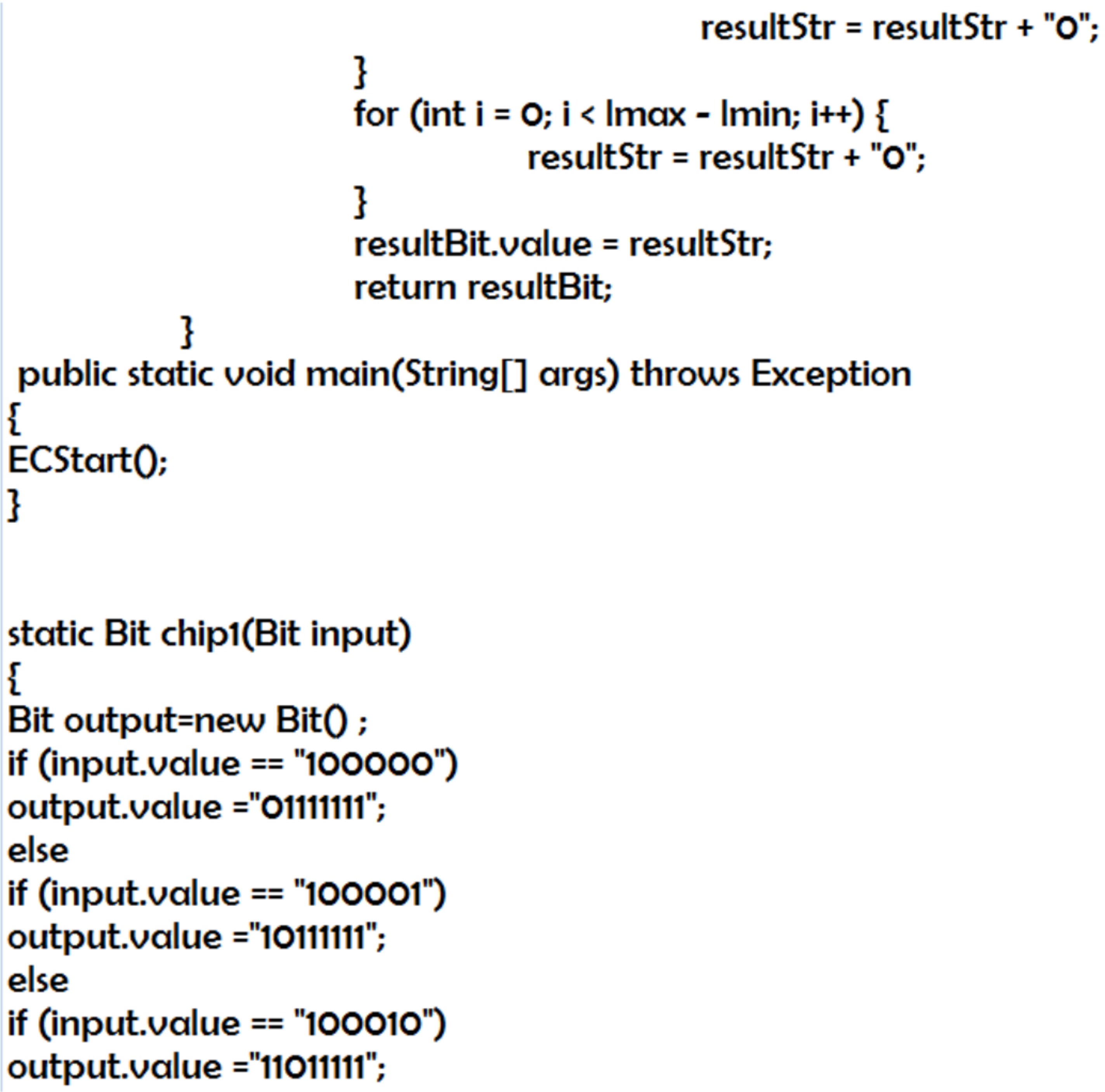


74HC138.ec → 74HC138.java

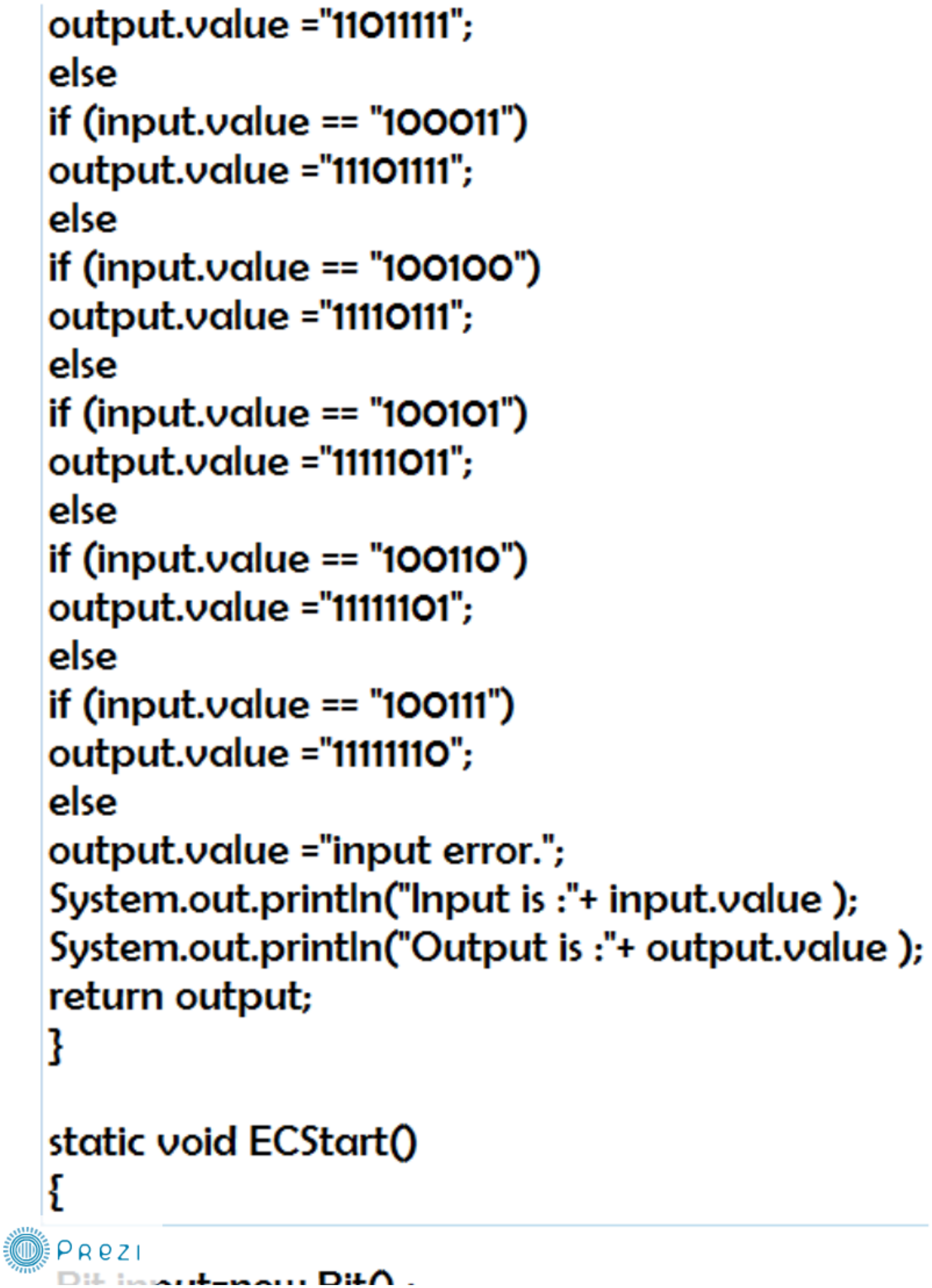
```
bit main(){
bit input;
bit output;
input.value= "100000"b;
return output=chip1(input);
}
bit chip1(bit input){
bit output;
if          (input.value          ==          "100000"b)          output.value="01111111"b;
else if     (input.value          ==          "100001"b)          output.value="10111111"b;
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;

}
```

```
package ec;

public class test_chip1_result
{   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;
        }
    }
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++) {
```

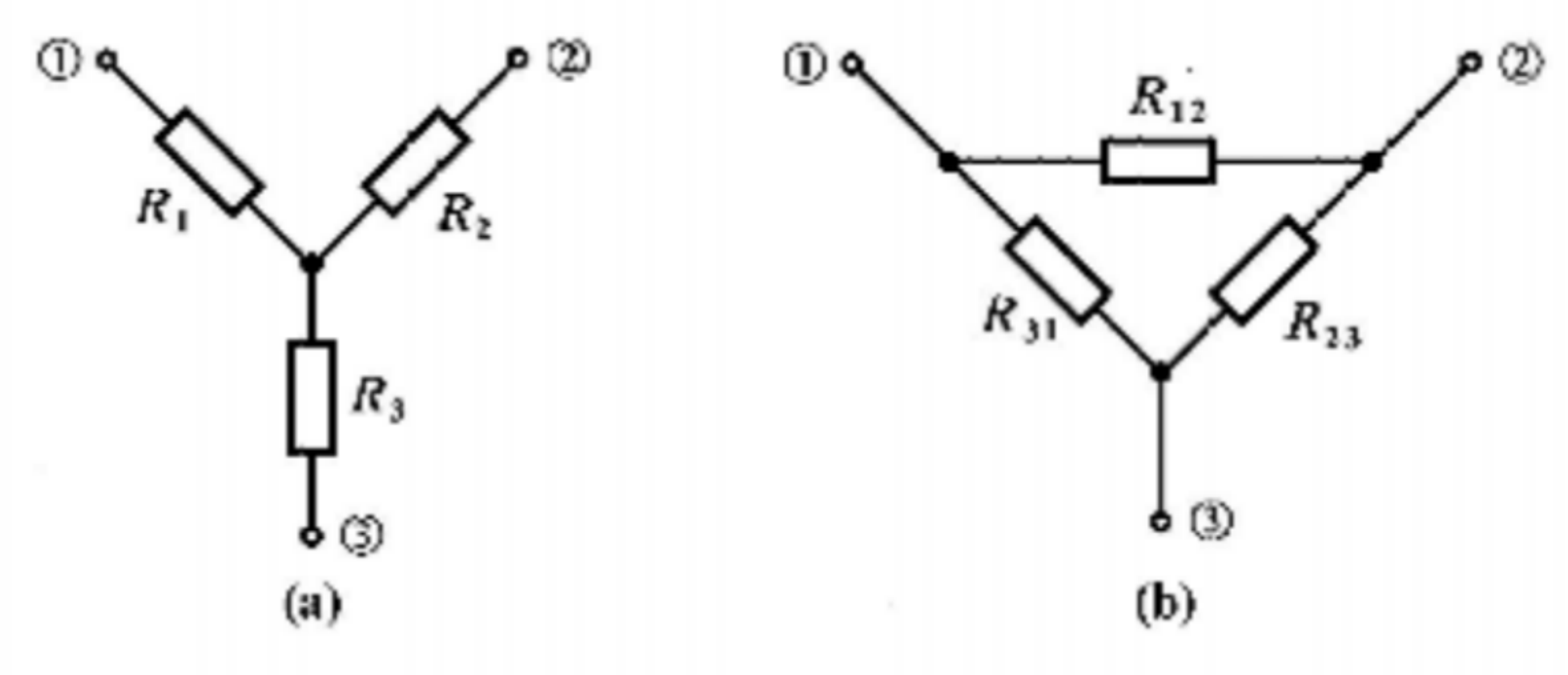
```
        if (result[i])
            resultStr = resultStr + "1";
        else
            resultStr = resultStr + "0";
    }
    for (int i = 0; i < lmax - lmin; i++) {
        resultStr = resultStr + "0";
    }
    resultBit.value = resultStr;
    return resultBit;
}public static Bit aORb(Bit a, 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";
```



```
                resultStr = resultStr + "0";
        }
        for (int i = 0; i < lmax - lmin; i++) {
            resultStr = resultStr + "0";
        }
        resultBit.value = resultStr;
        return resultBit;
    }
 public static void main(String[] args) throws Exception
{
ECStart();
}


static Bit chip1(Bit input)
{
Bit output=new Bit() ;
if (input.value == "100000")
output.value ="01111111";
else
if (input.value == "100001")
output.value ="10111111";
else
if (input.value == "100010")
output.value ="11011111";
```

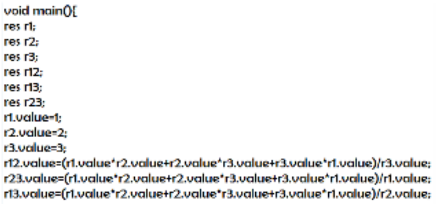


```
output.value ="11011111";
else
if (input.value == "100011")
output.value ="11101111";
else
if (input.value == "100100")
output.value ="11110111";
else
if (input.value == "100101")
output.value ="11111011";
else
if (input.value == "100110")
output.value ="11111101";
else
if (input.value == "100111")
output.value ="11111110";
else
output.value ="input error.";
System.out.println("Input is :"+ input.value );
System.out.println("Output is :"+ output.value );
return output;
}

static void ECStart()
{
Bit input=new Bit() ;
Bit input=new Bit() ;
Bit output=new Bit() ;
input.value ="100000";
output=chip1(input);
}

}
```

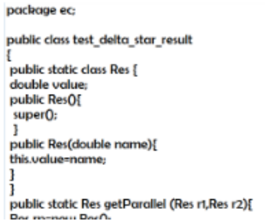


y configuration(left) and delta configuration(right)

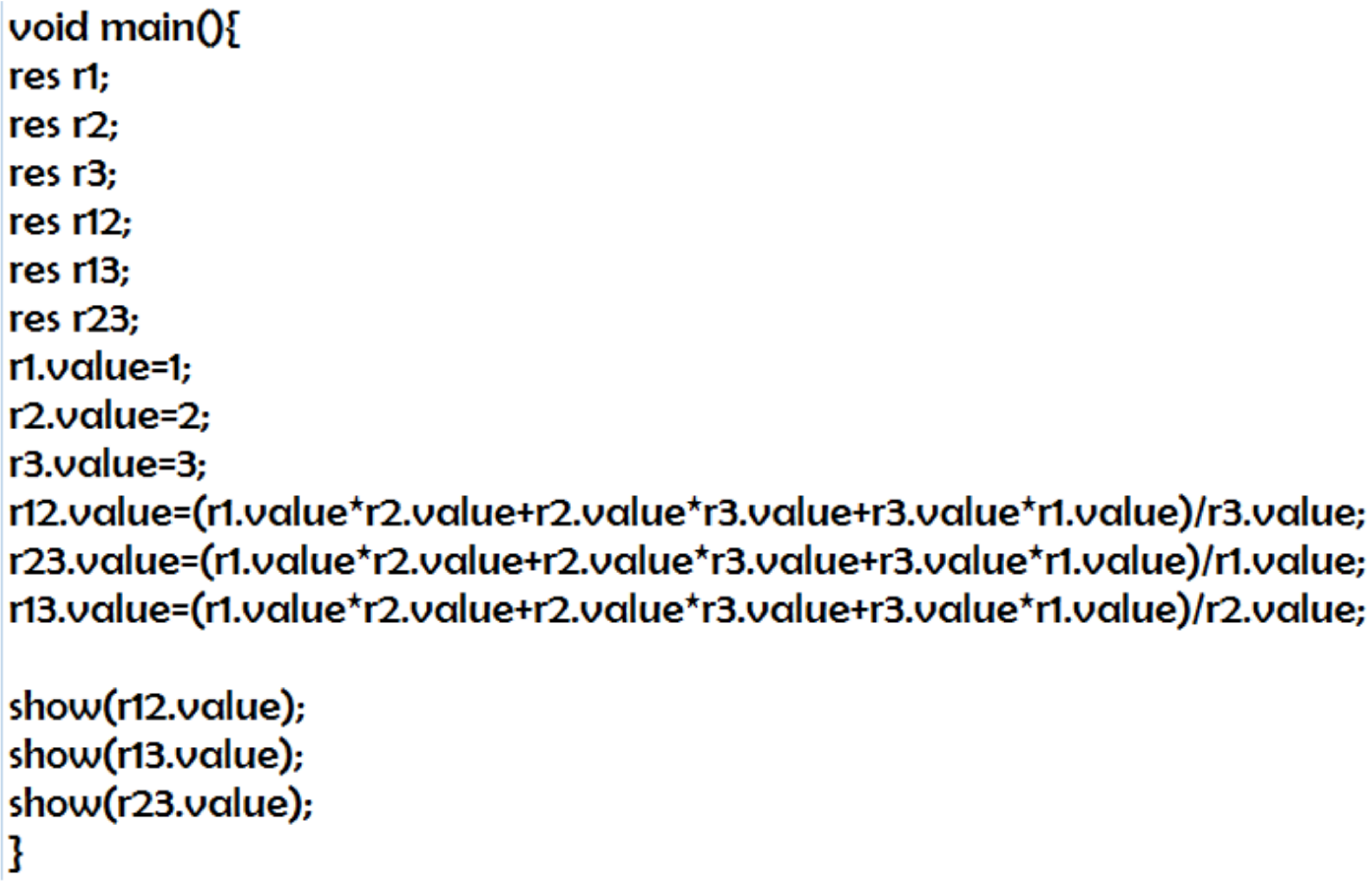
delta_star.ec → delta_star.java



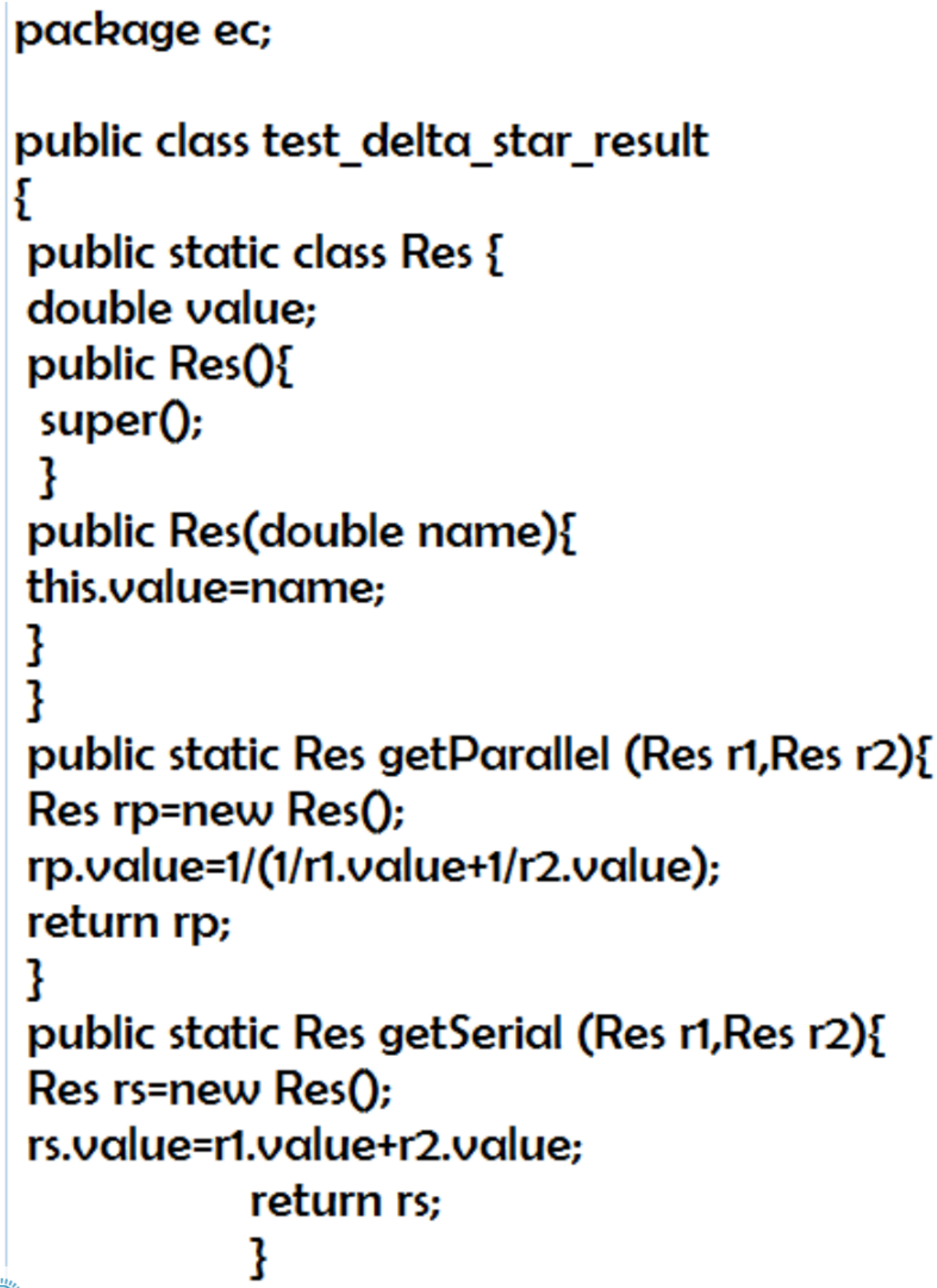
```
void main(){
res r1;
res r2;
res r3;
res r12;
res r13;
res r23;
r1.value=1;
r2.value=2;
r3.value=3;
r12.value=(r1.value*r2.value+r2.value*r3.value+r3.value*r1.value)/r3.value;
r23.value=(r1.value*r2.value+r2.value*r3.value+r3.value*r1.value)/r1.value;
r13.value=(r1.value*r2.value+r2.value*r3.value+r3.value*r1.value)/r2.value;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



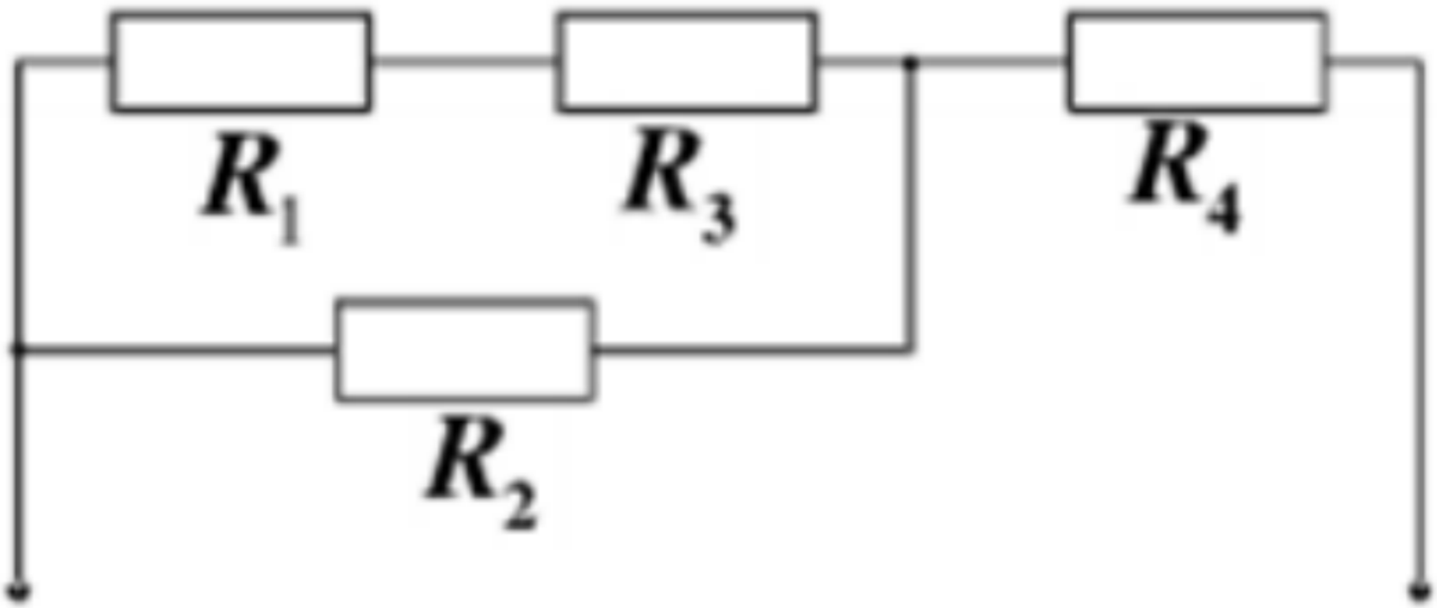
```
package ec;

public class test_delta_star_result
{
 public static class Res {
 double value;
 public Res(){
  super();
  }
 public Res(double name){
 this.value=name;
 }
 }
 public static Res getParallel (Res r1,Res r2){
 Res rn=new Res();
```



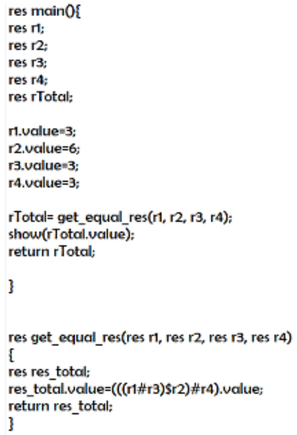
```
void main(){
res r1;
res r2;
res r3;
res r12;
res r13;
res r23;
r1.value=1;
r2.value=2;
r3.value=3;
r12.value=(r1.value*r2.value+r2.value*r3.value+r3.value*r1.value)/r3.value;
r23.value=(r1.value*r2.value+r2.value*r3.value+r3.value*r1.value)/r1.value;
r13.value=(r1.value*r2.value+r2.value*r3.value+r3.value*r1.value)/r2.value;

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



```
package ec;

public class test_delta_star_result
{
 public static class Res {
 double value;
 public Res(){
  super();
  }
 public Res(double name){
 this.value=name;
 }
 }
 public static Res getParallel (Res r1,Res r2){
 Res rp=new Res();
 rp.value=1/(1/r1.value+1/r2.value);
 return rp;
 }
 public static Res getSerial (Res r1,Res r2){
 Res rs=new Res();
 rs.value=r1.value+r2.value;
                  return rs;
                  }
 public static void main(String[] args) throws Exception
{
ECStart();
}
static void ECStart()
{
Res r1=new Res() ;
Res r2=new Res() ;
Res r3=new Res() ;
Res r12=new Res() ;
Res r13=new Res() ;
Res r23=new Res() ;
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 ;
System.out.println(r12.value );
System.out.println(r13.value );
System.out.println(r23.value );
}


}
```

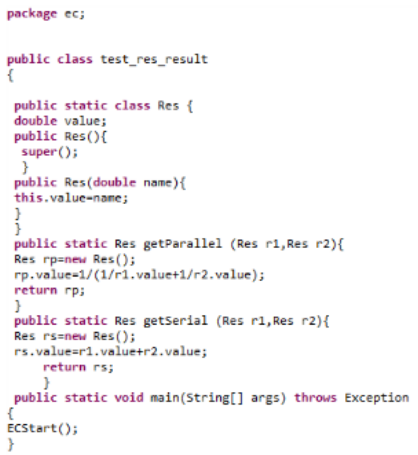


get_equal_res.ec

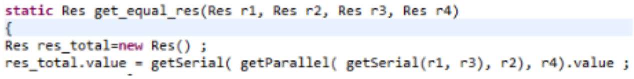


```
res main(){
res r1;
res r2;
res r3;
res r4;
res rTotal;

r1.value=3;
r2.value=6;
r3.value=3;
r4.value=3;

rTotal= get_equal_res(r1, r2, r3, r4);
show(rTotal.value);
return rTotal;

}


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;
}
```

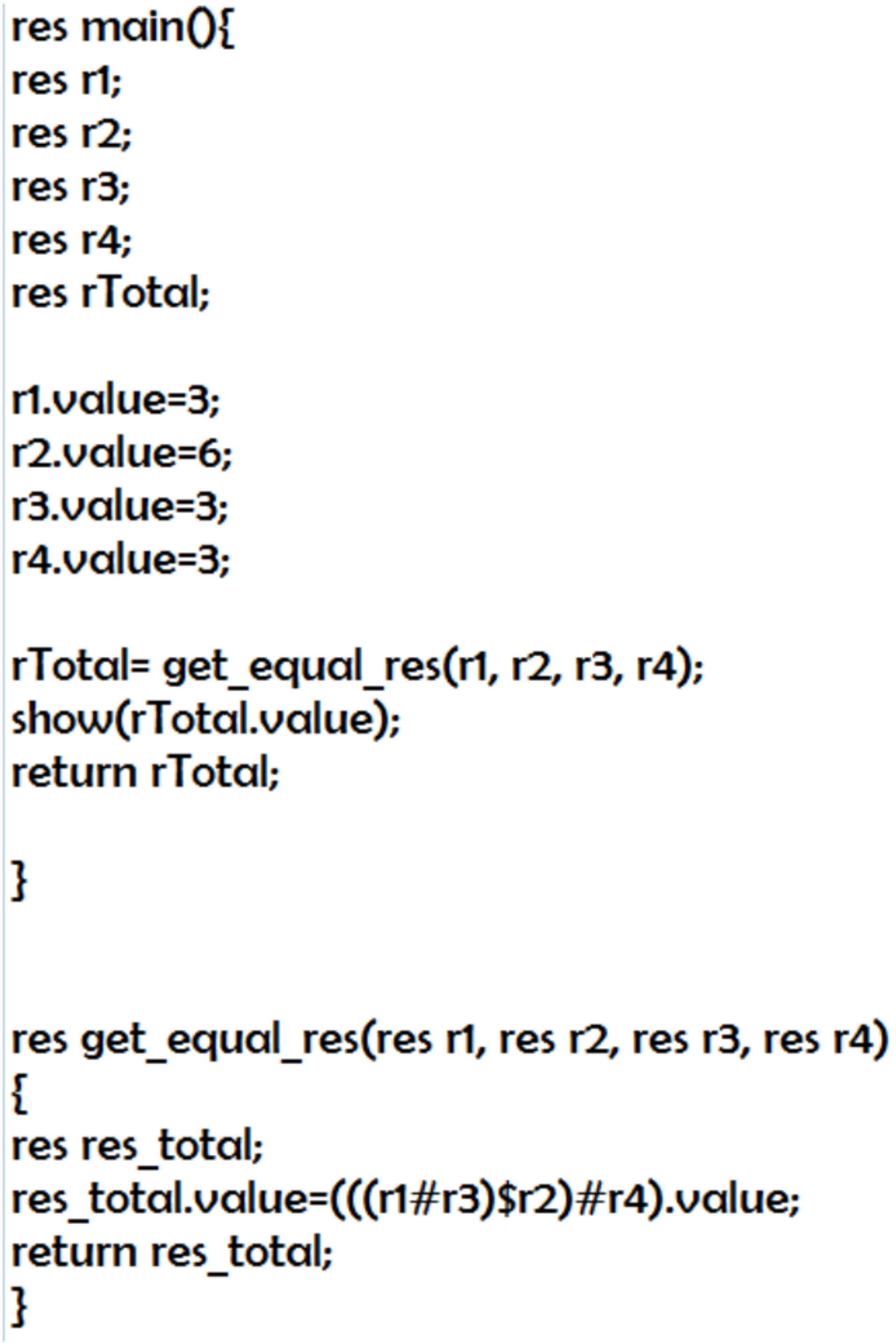
get_equal_res.java



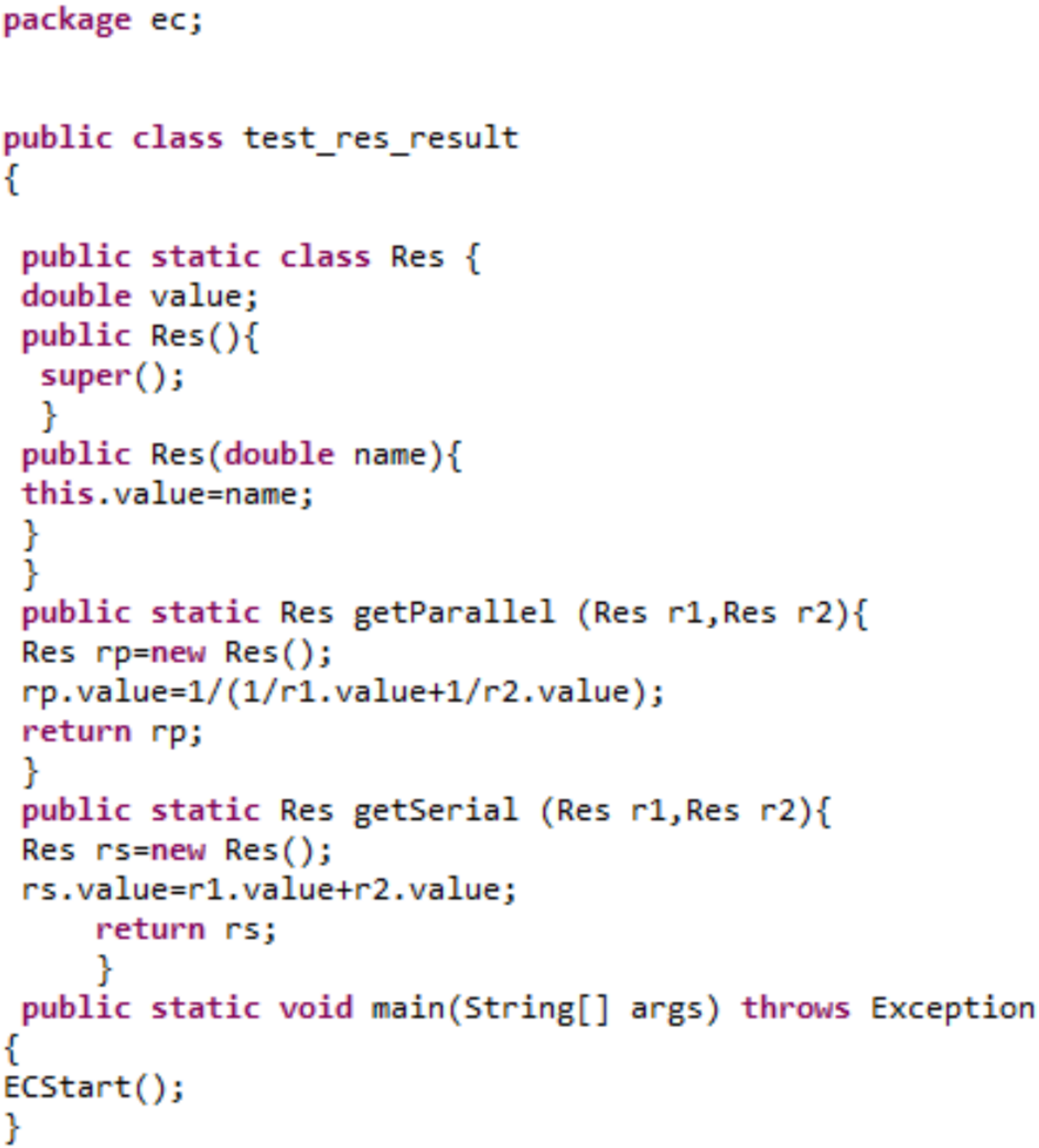
```
package ec;


public class test_res_result
{

 public static class Res {
 double value;
 public Res(){
  super();
  }
 public Res(double name){
 this.value=name;
 }
 }
 public static Res getParallel (Res r1,Res r2){
 Res rp=new Res();
 rp.value=1/(1/r1.value+1/r2.value);
 return rp;
 }
 public static Res getSerial (Res r1,Res r2){
 Res rs=new Res();
 rs.value=r1.value+r2.value;
     return rs;
     }
 public static void main(String[] args) throws Exception
{
ECStart();
}
```



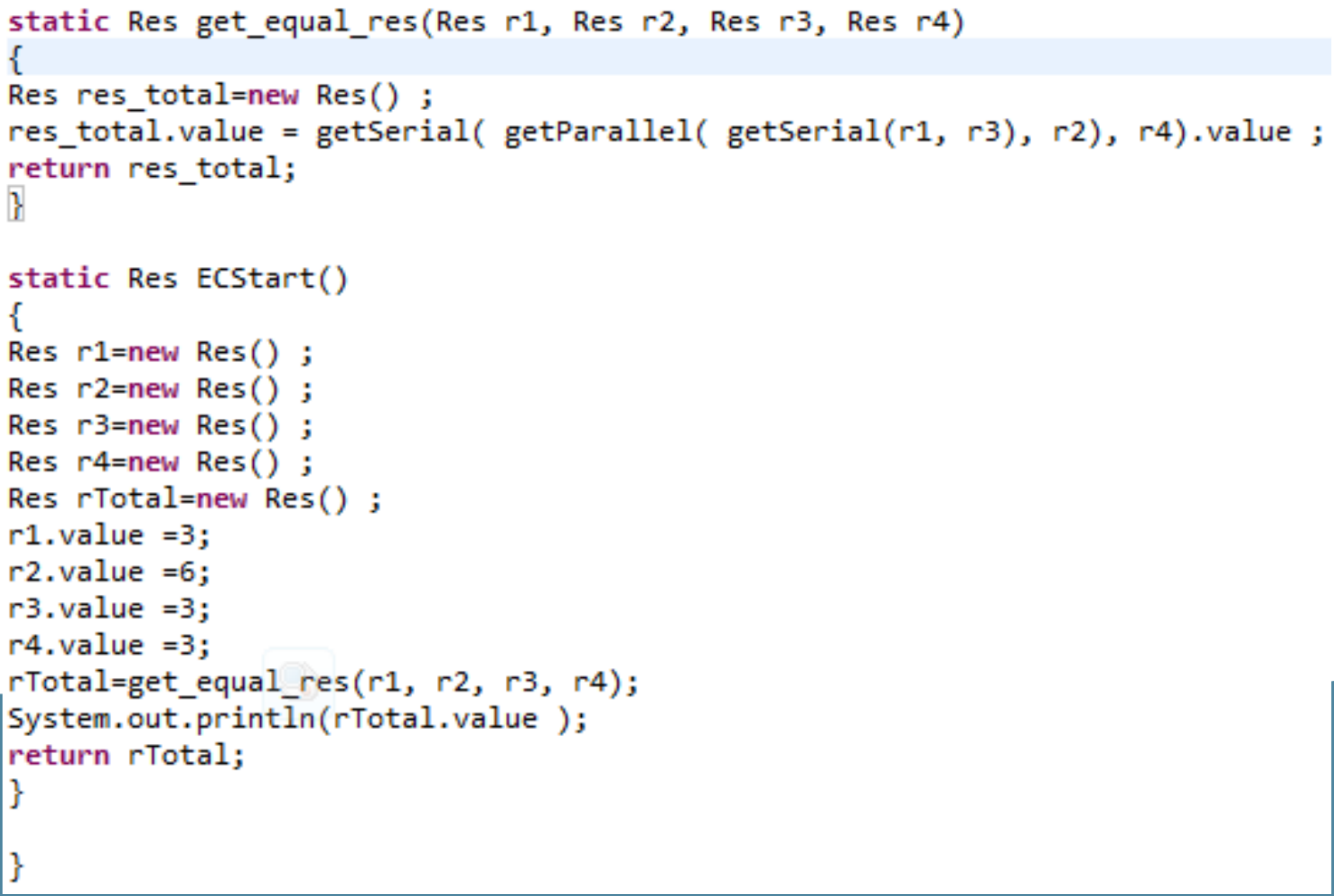
```
static Res get_equal_res(Res r1, Res r2, Res r3, Res r4)
{
Res res_total=new Res() ;
res_total.value = getSerial( getParallel( getSerial(r1, r3), r2), r4).value ;
```



```
res main(){
res r1;
res r2;
res r3;
res r4;
res rTotal;

r1.value=3;
r2.value=6;
r3.value=3;
r4.value=3;

rTotal= get_equal_res(r1, r2, r3, r4);
show(rTotal.value);
return rTotal;

}


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;
}
```

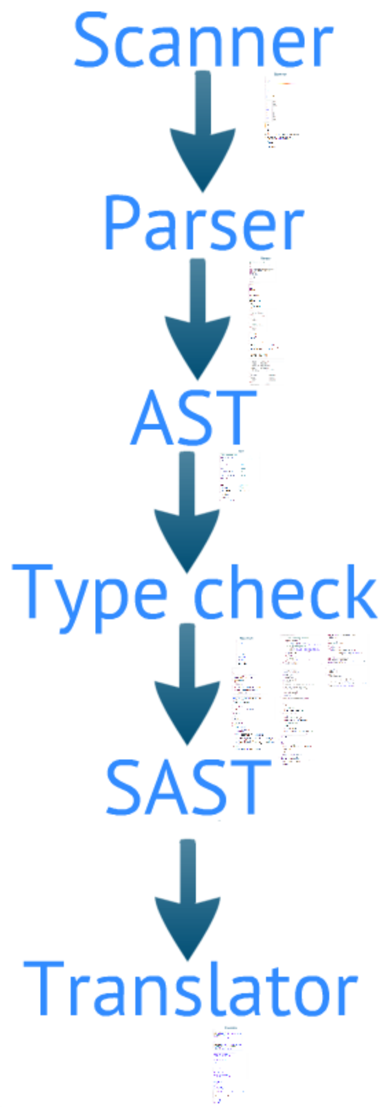


```
package ec;


public class test_res_result
{

 public static class Res {
 double value;
 public Res(){
  super();
  }
 public Res(double name){
 this.value=name;
 }
 }
 public static Res getParallel (Res r1,Res r2){
 Res rp=new Res();
 rp.value=1/(1/r1.value+1/r2.value);
 return rp;
 }
 public static Res getSerial (Res r1,Res r2){
 Res rs=new Res();
 rs.value=r1.value+r2.value;
     return rs;
     }
 public static void main(String[] args) throws Exception
{
ECStart();
}
```



```
static Res get_equal_res(Res r1, Res r2, Res r3, Res r4)
{
Res res_total=new Res() ;
res_total.value = getSerial( getParallel( getSerial(r1, r3), r2), r4).value ;
return res_total;
}

static Res ECStart()
{
Res r1=new Res() ;
Res r2=new Res() ;
Res r3=new Res() ;
Res r4=new Res() ;
Res rTotal=new Res() ;
r1.value =3;
r2.value =6;
r3.value =3;
r4.value =3;
rTotal=get_equal_res(r1, r2, r3, r4);
System.out.println(rTotal.value );
return rTotal;
}

}
```

Compiler Architecture



Scanner

Parser

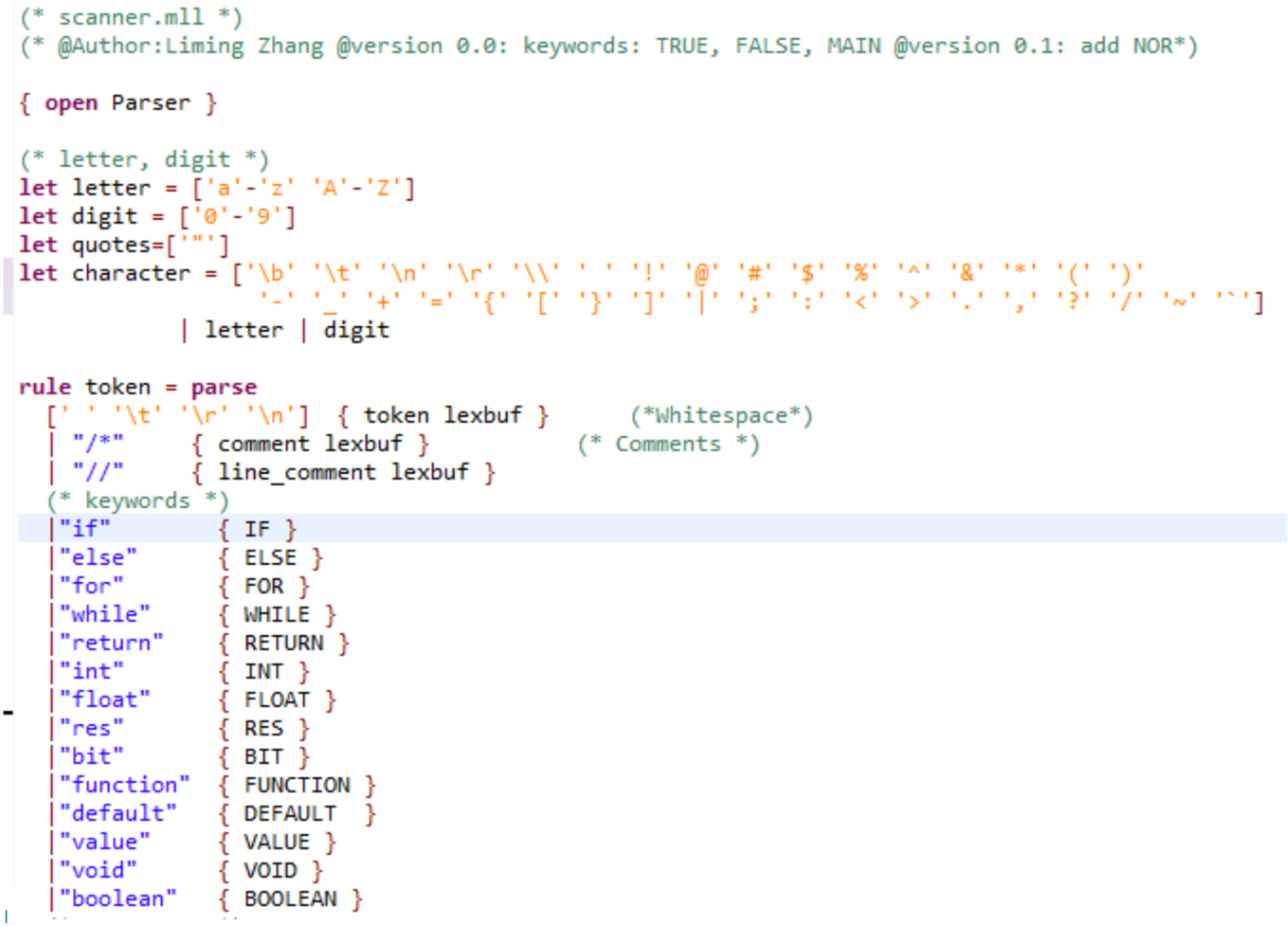
AST

Type check

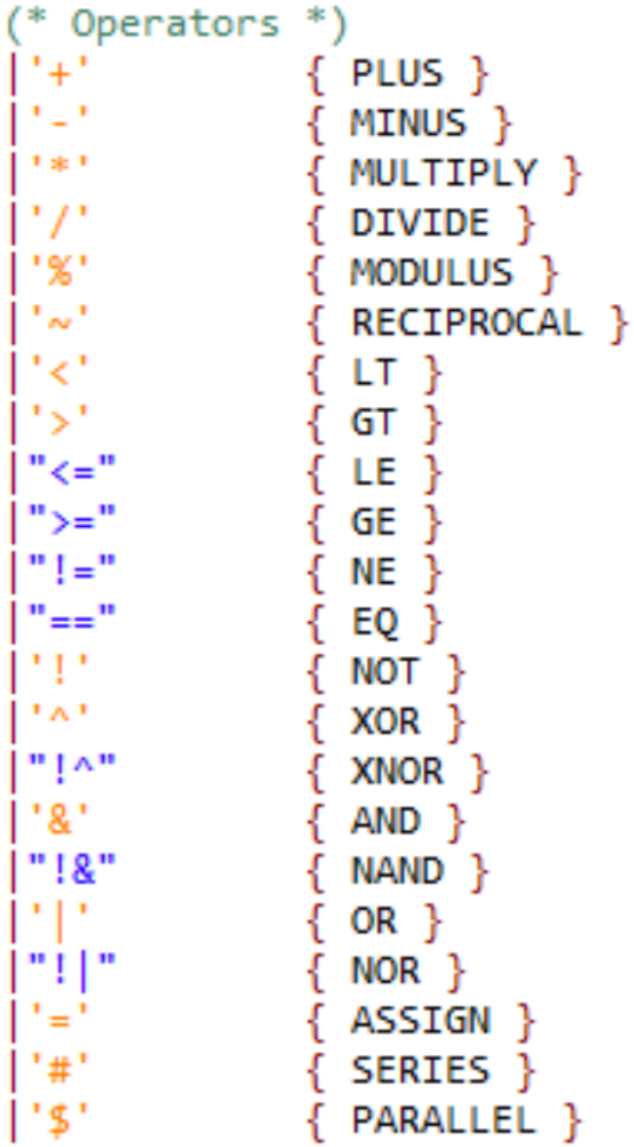
SAST

Translator

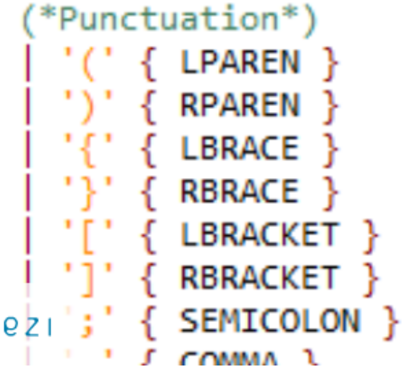
Scanner



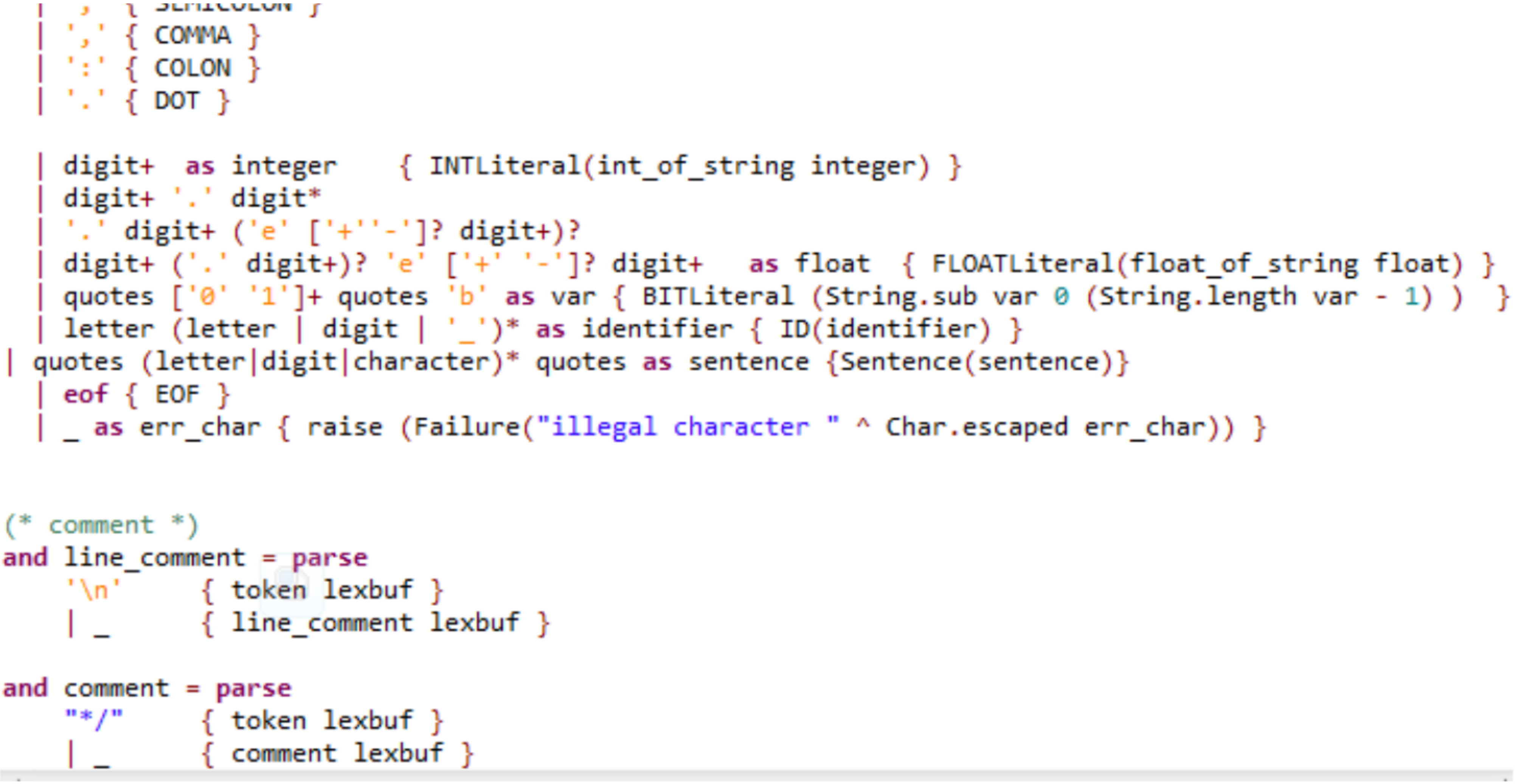
```
(* scanner.mll *)
(* @Author:Liming Zhang @version 0.0: keywords: TRUE, FALSE, MAIN @version 0.1: add NOR*)

{ 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 }
  |"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 }
```

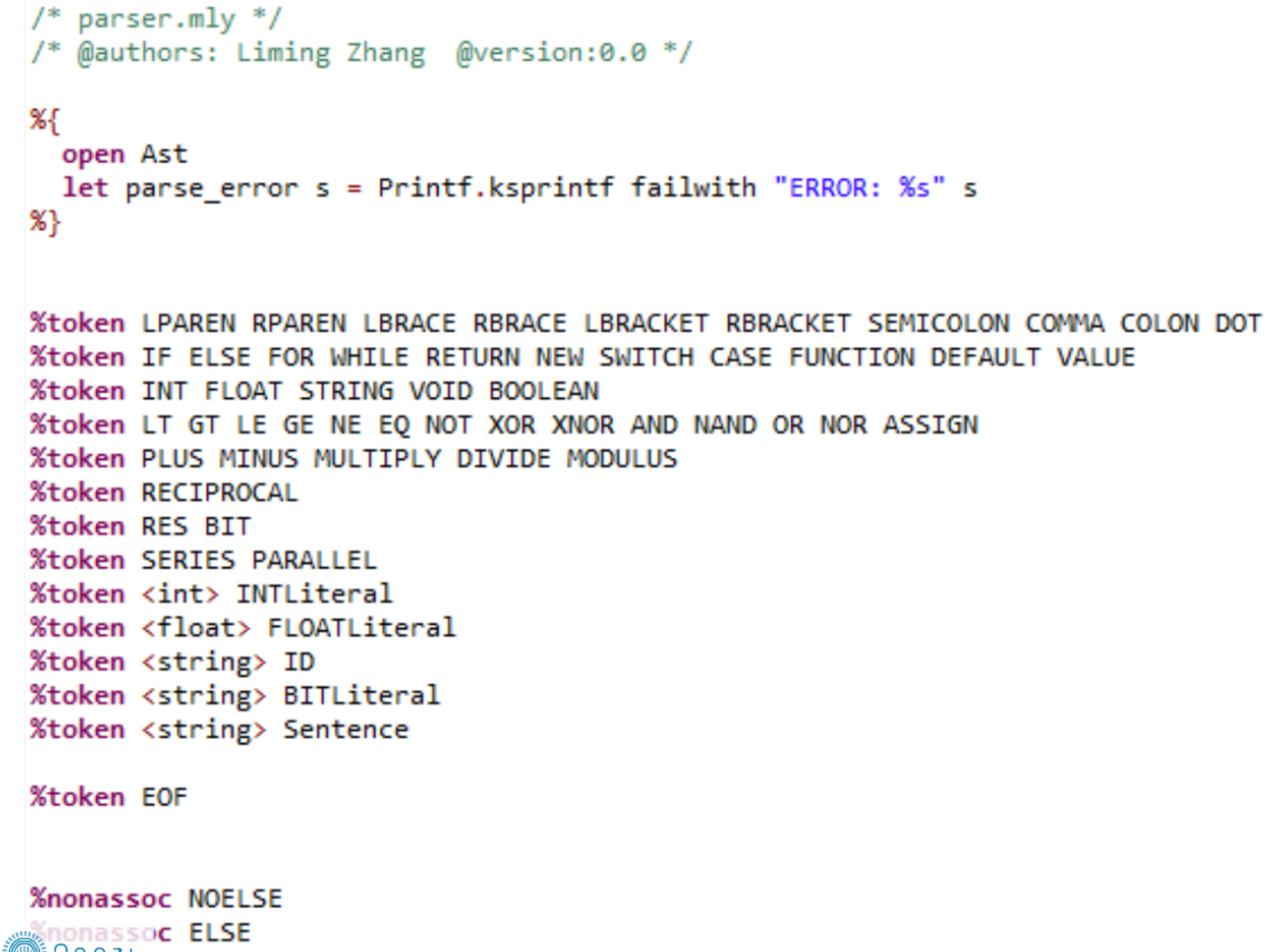


```
(*Punctuation*)
| '(' { LPAREN }
| ')' { RPAREN }
| '{' { LBRACE }
| '}' { RBRACE }
| '[' { LBRACKET }
| ']' { RBRACKET }
| ';' { SEMICOLON }
| ',' { COMMA }
```

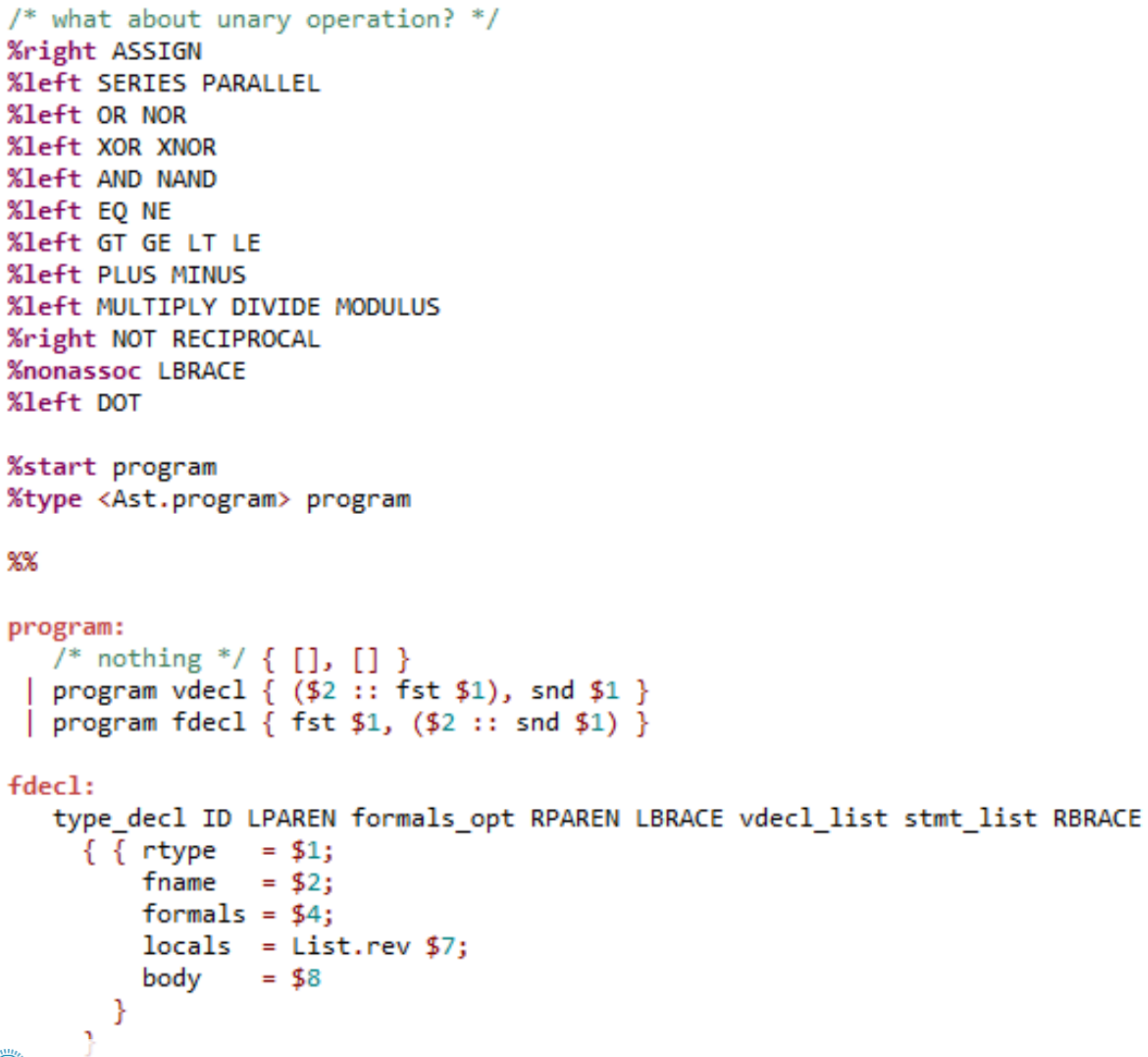


```
    | ';' { SEMICOLON }
    | ',' { COMMA }
    | ':' { COLON }
    | '.' { DOT }

    | 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 }
```

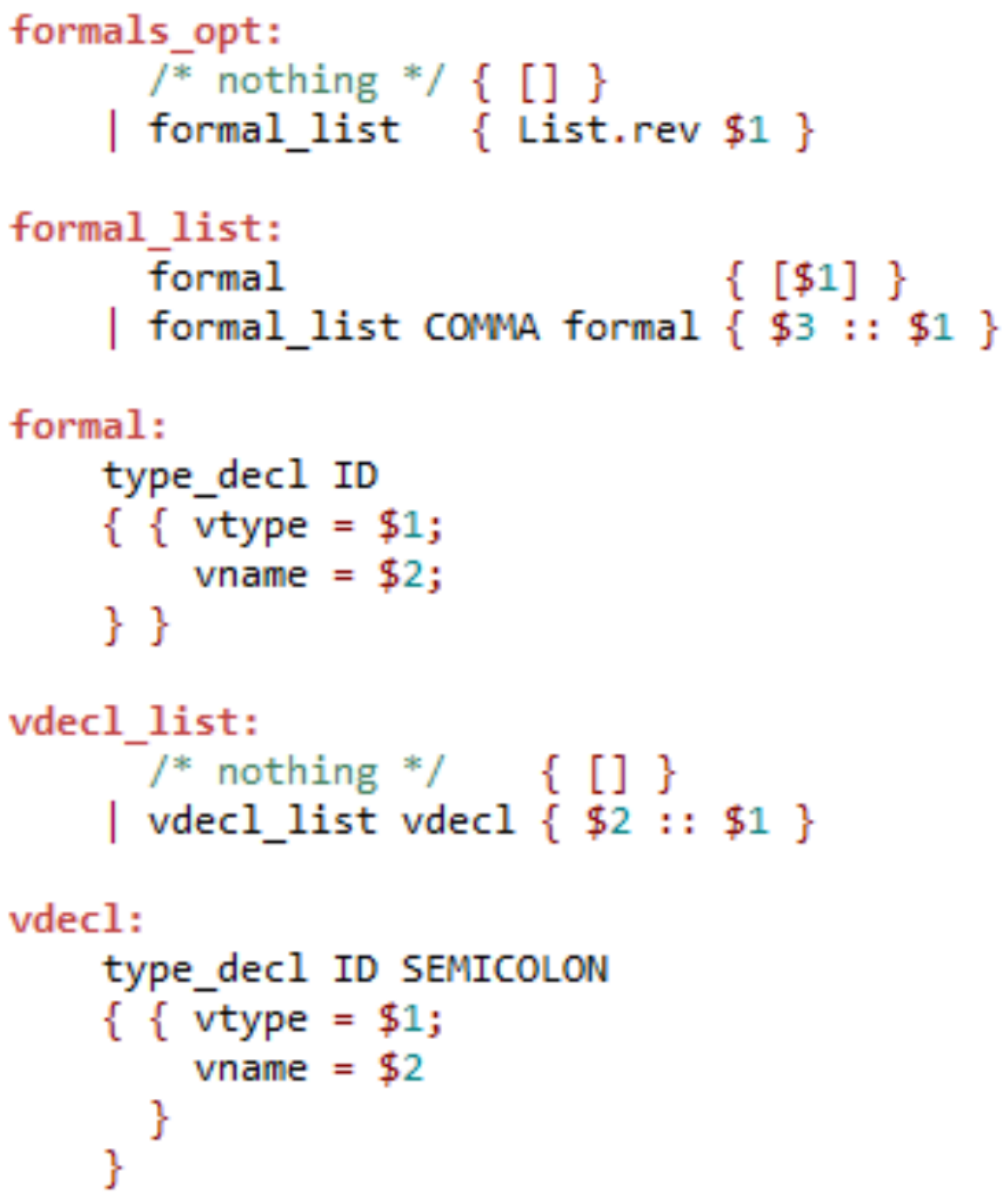
Parser



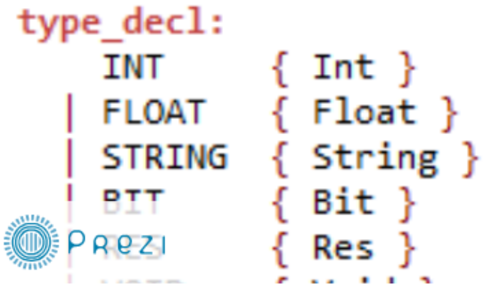
```
/* parser.mly */
/* @authors: Liming Zhang  @version:0.0 */

%{
  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
```



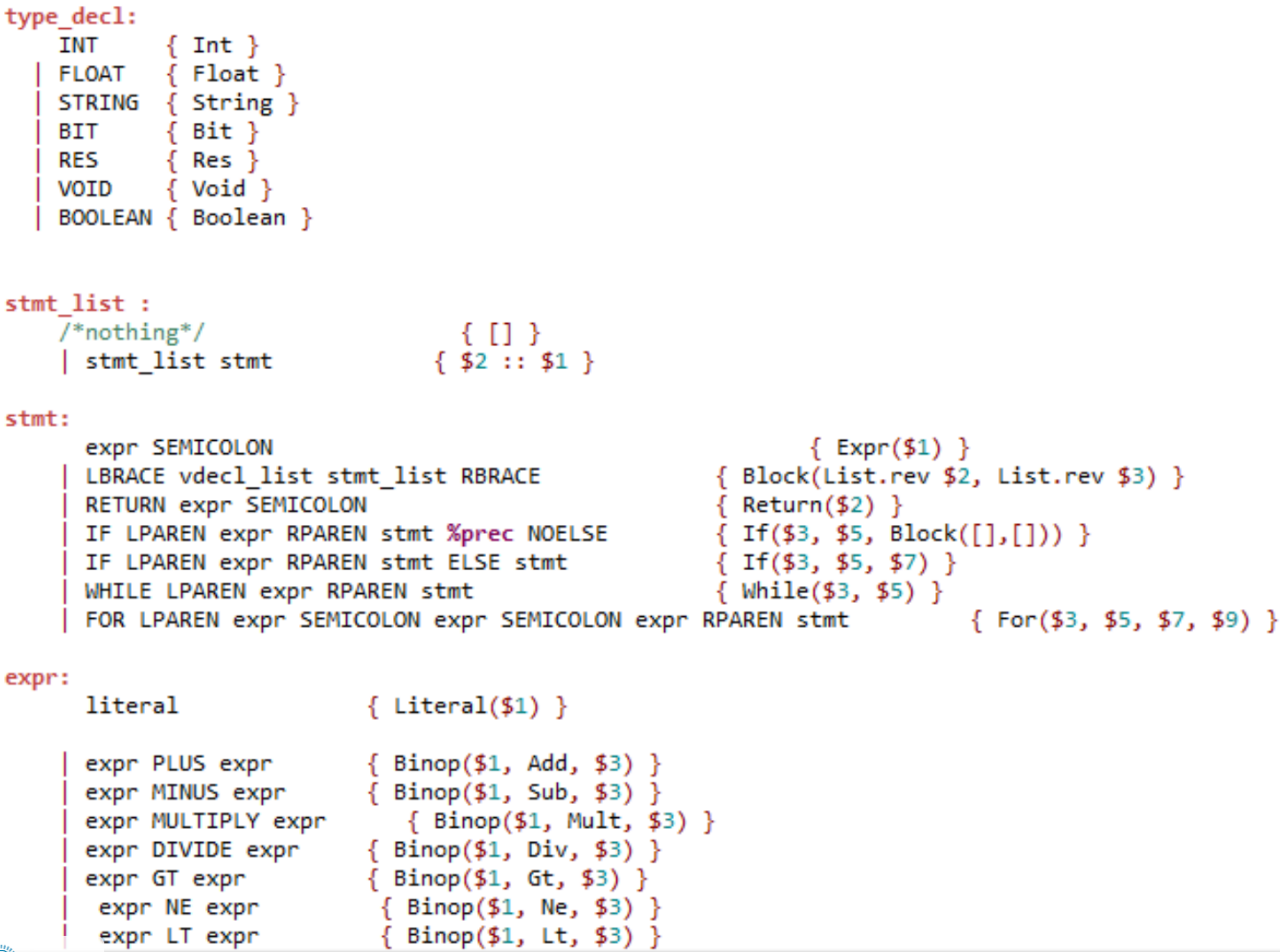
```
/* what about unary operation? */
%right ASSIGN
%left SERIES PARALLEL
%left OR NOR
%left XOR XNOR
%left AND NAND
%left EQ NE
%left GT GE LT LE
%left PLUS MINUS
%left MULTIPLY DIVIDE MODULUS
%right NOT RECIPROCAL
%nonassoc LBRACE
%left DOT

%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
       }
     }
```



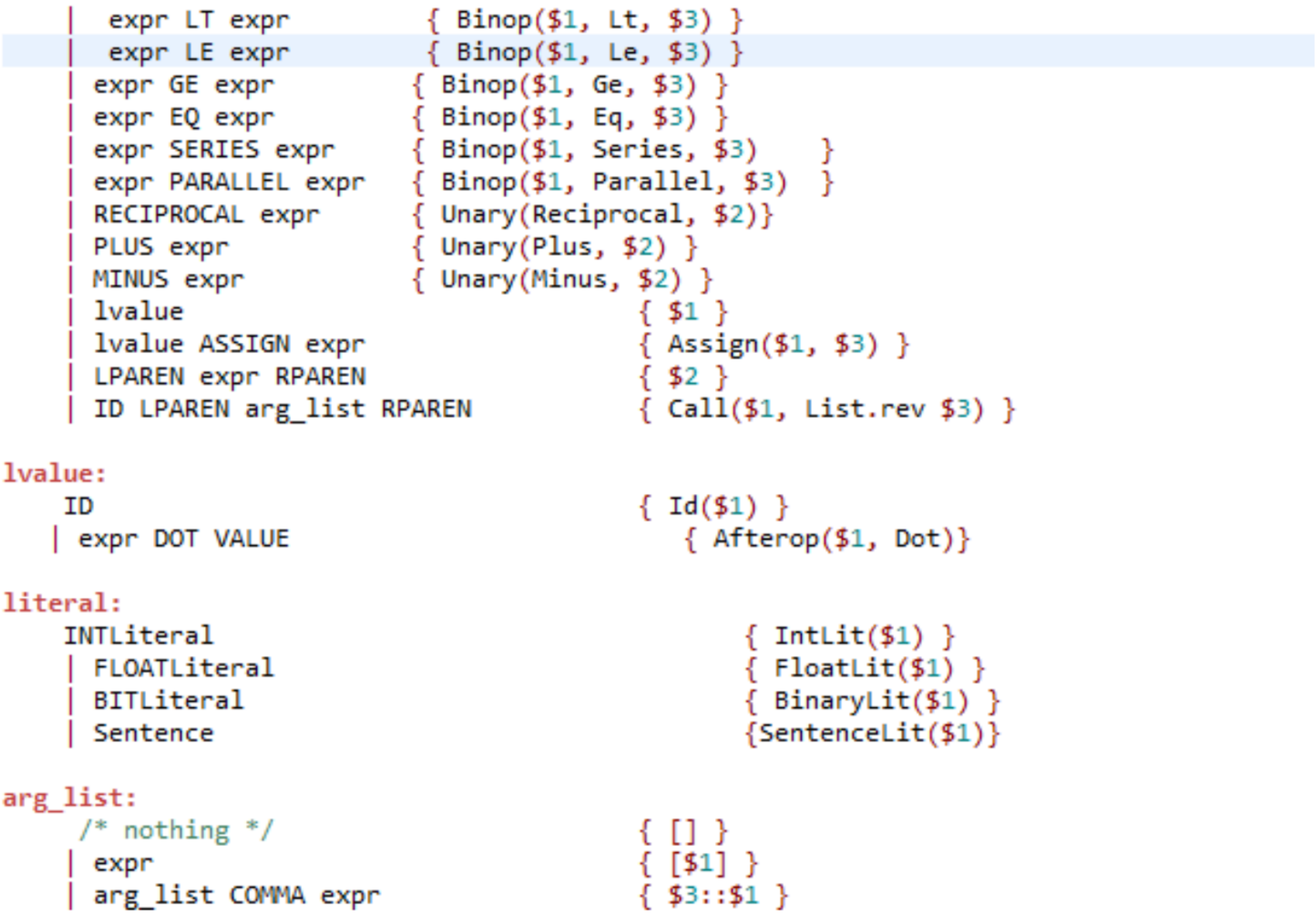
```
formals_opt:
      /* nothing */ { [] }
    | formal_list   { List.rev $1 }

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 }
```

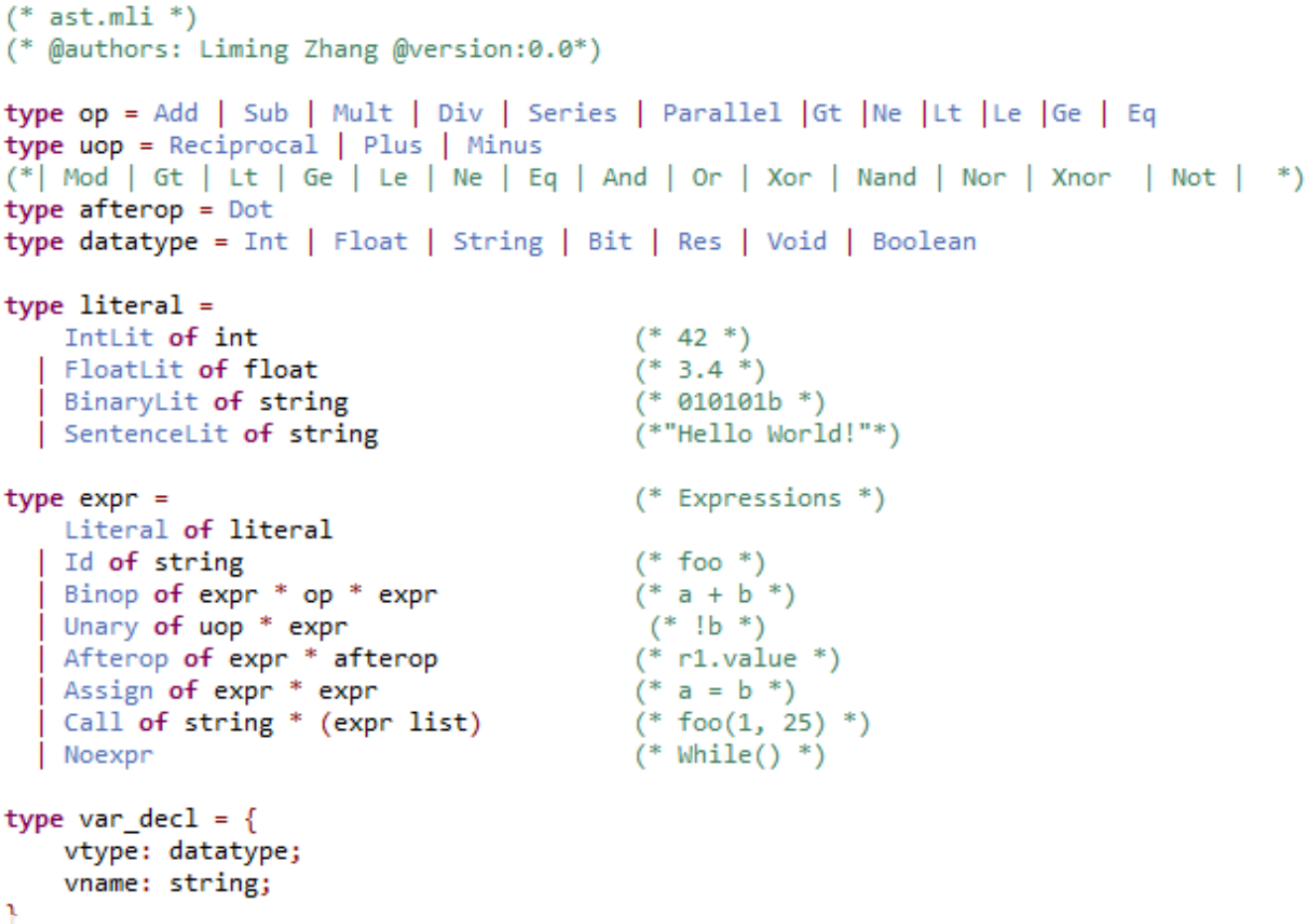


```
type_decl:
    INT     { Int }
  | FLOAT   { Float }
  | STRING  { String }
  | BIT     { Bit }
  | RES     { Res }
  | VOID    { Void }
  | BOOLEAN { Boolean }


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) }
```

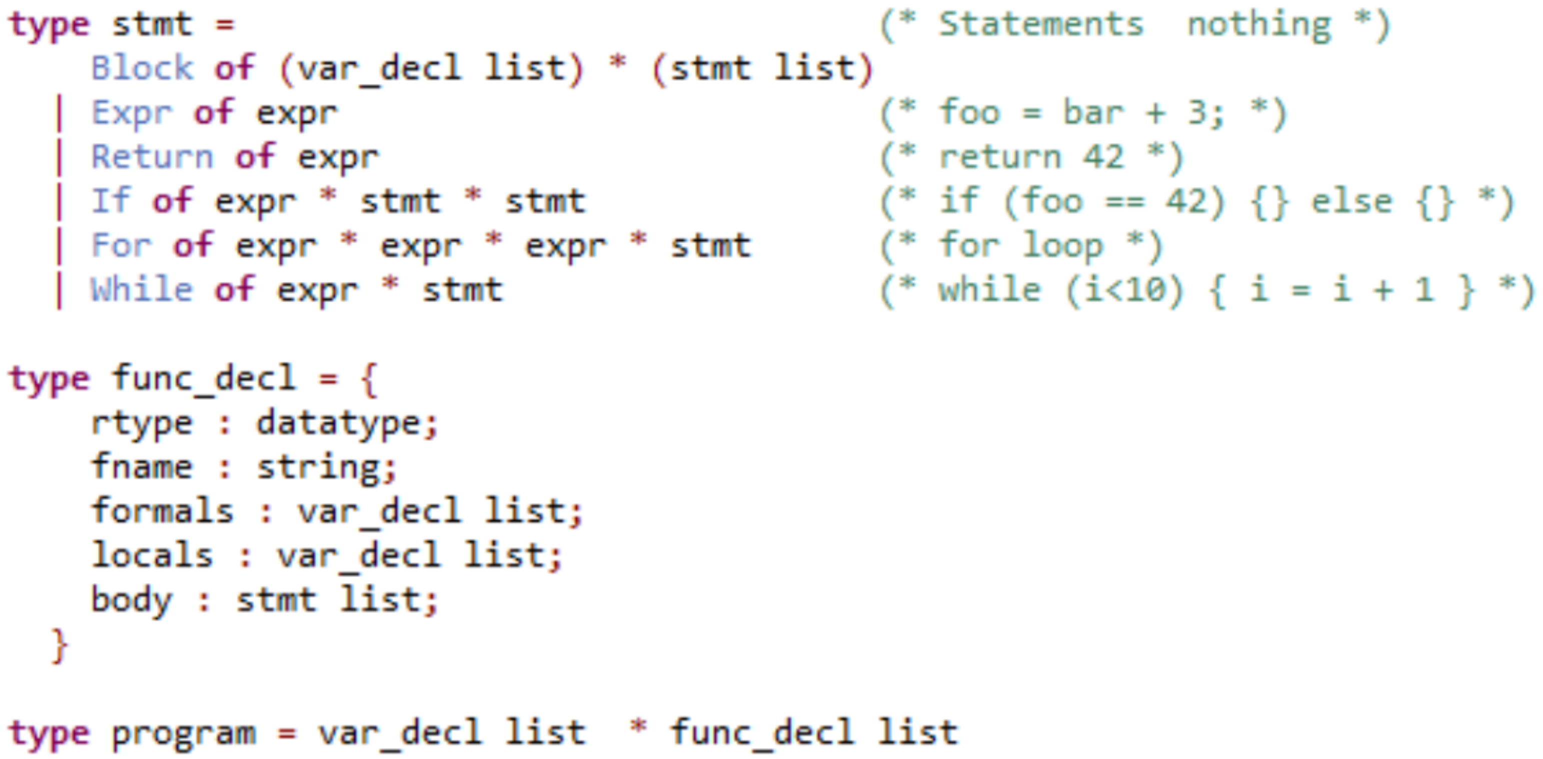


```
    |  expr LT expr         { Binop($1, Lt, $3) }
    |  expr LE expr         { Binop($1, Le, $3) }
    | expr GE expr         { Binop($1, Ge, $3) }
    | expr EQ expr         { Binop($1, Eq, $3) }
    | expr SERIES expr     { Binop($1, Series, $3)    }
    | expr PARALLEL expr   { Binop($1, Parallel, $3)  }
    | 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

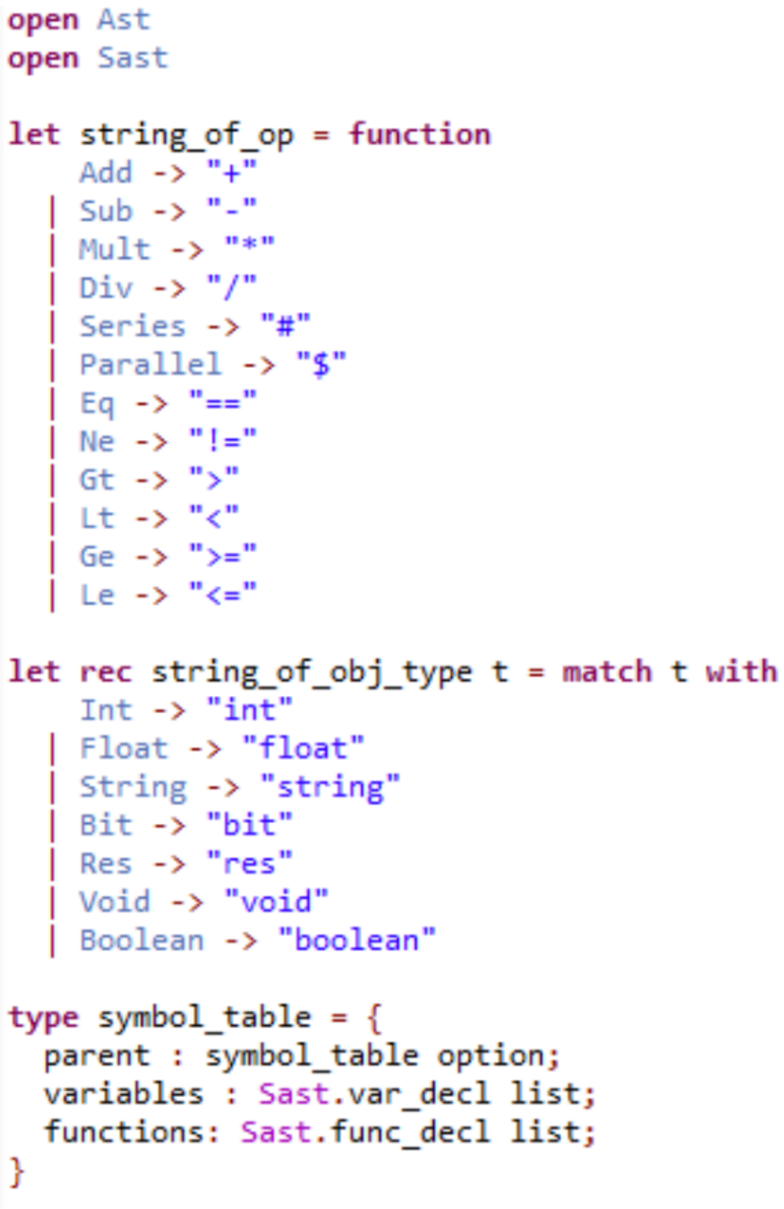


```
(* ast.mli *)
(* @authors: Liming Zhang @version:0.0*)

type op = Add | Sub | Mult | Div | Series | Parallel |Gt |Ne |Lt |Le |Ge | Eq
type uop = Reciprocal | Plus | Minus
(*| Mod | Gt | Lt | Ge | Le | Ne | Eq | And | Or | Xor | Nand | Nor | Xnor  | Not |   *)
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;
}
```

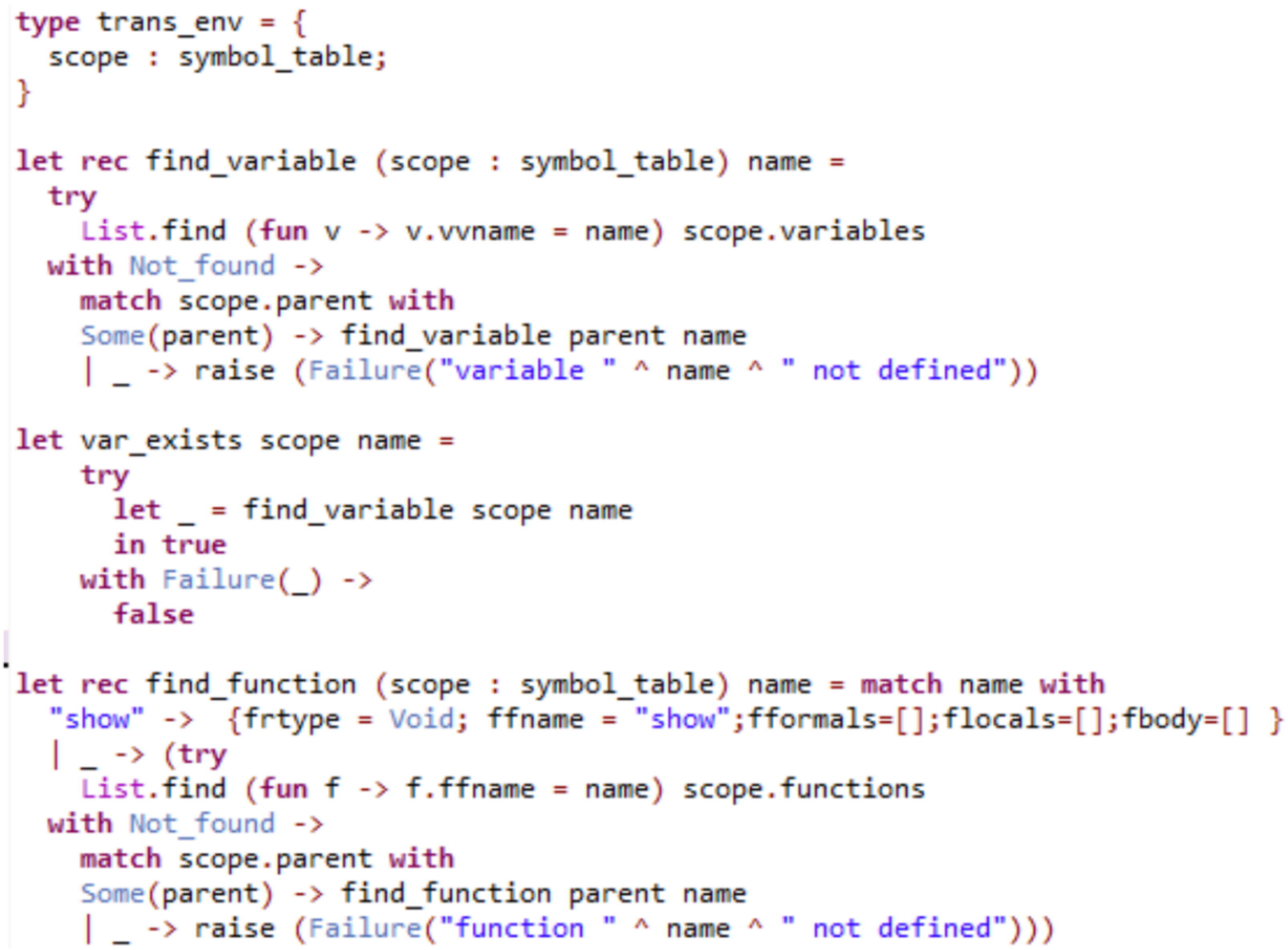


```
type stmt =                                   (* Statements  nothing *)
    Block of (var_decl list) * (stmt list)
  | Expr of expr                              (* foo = bar + 3; *)
  | Return of expr                            (* return 42 *)
  | If of expr * stmt * stmt                  (* if (foo == 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
```

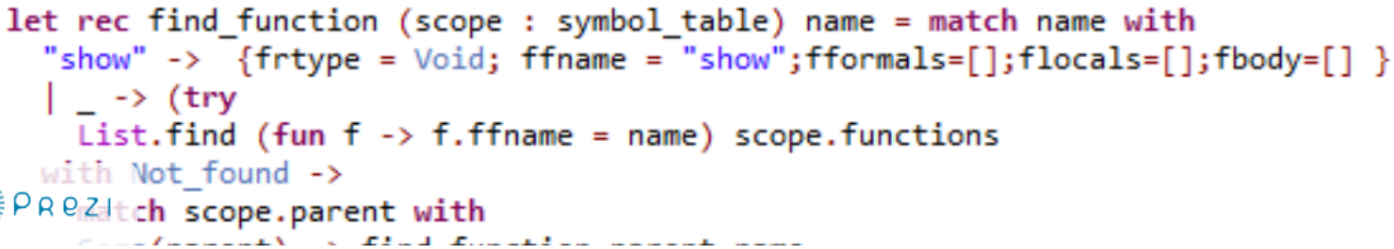
Type check



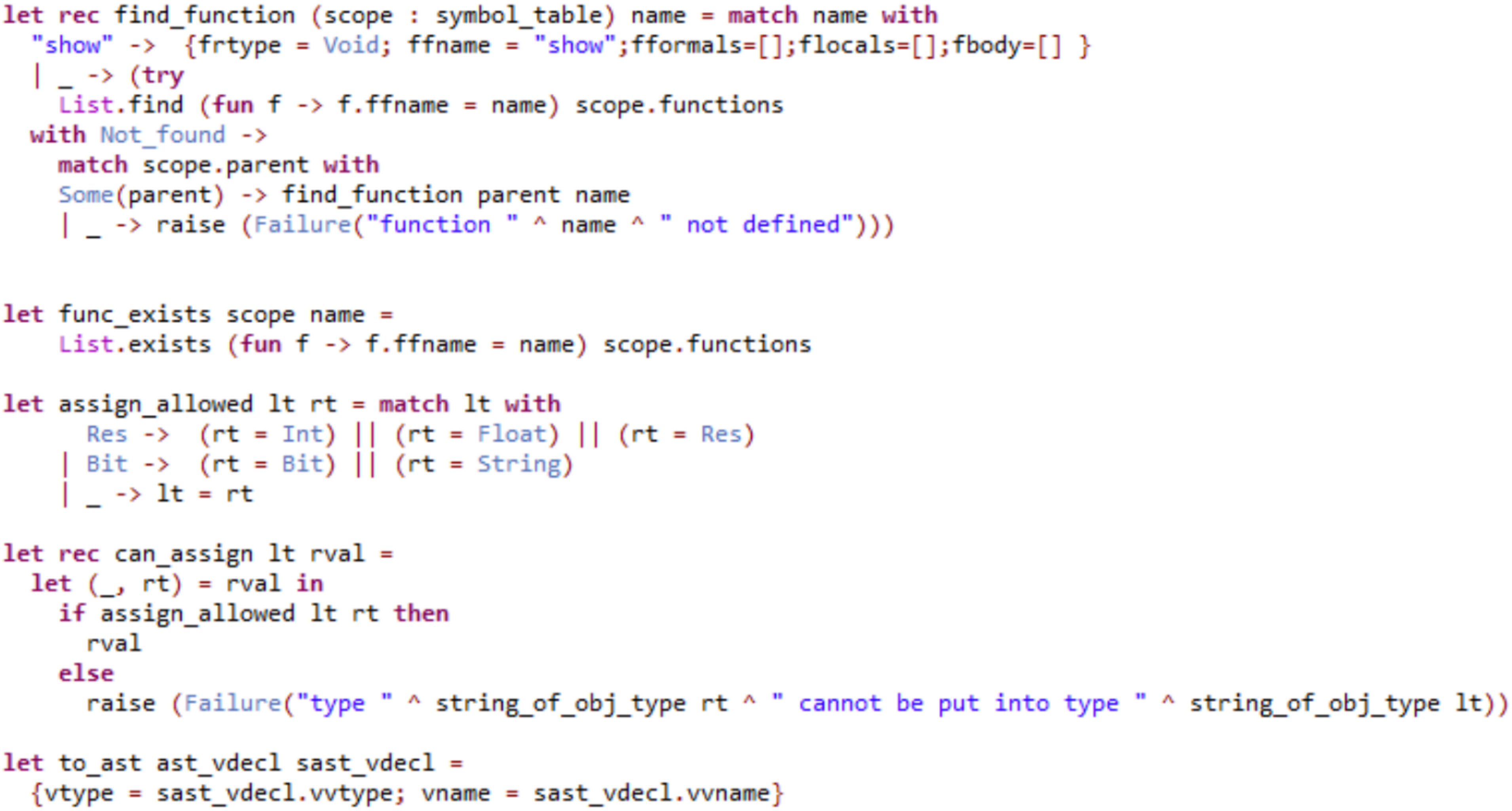
```
open Ast
open Sast

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;
}
```



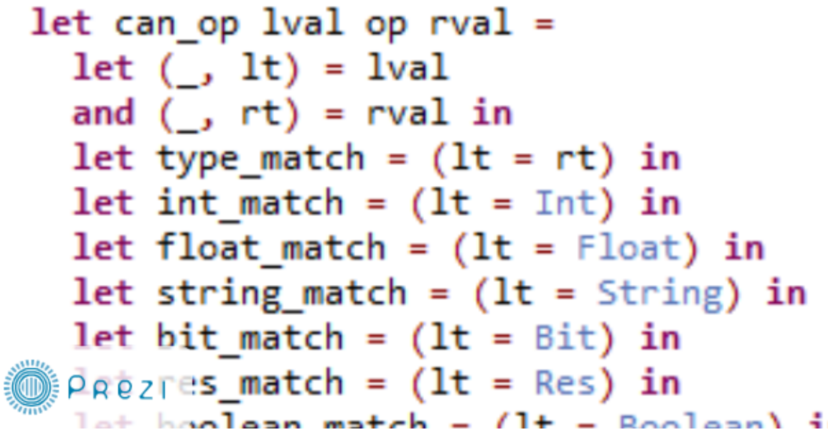
```
type trans_env = {
  scope : symbol_table;
}

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")))
```



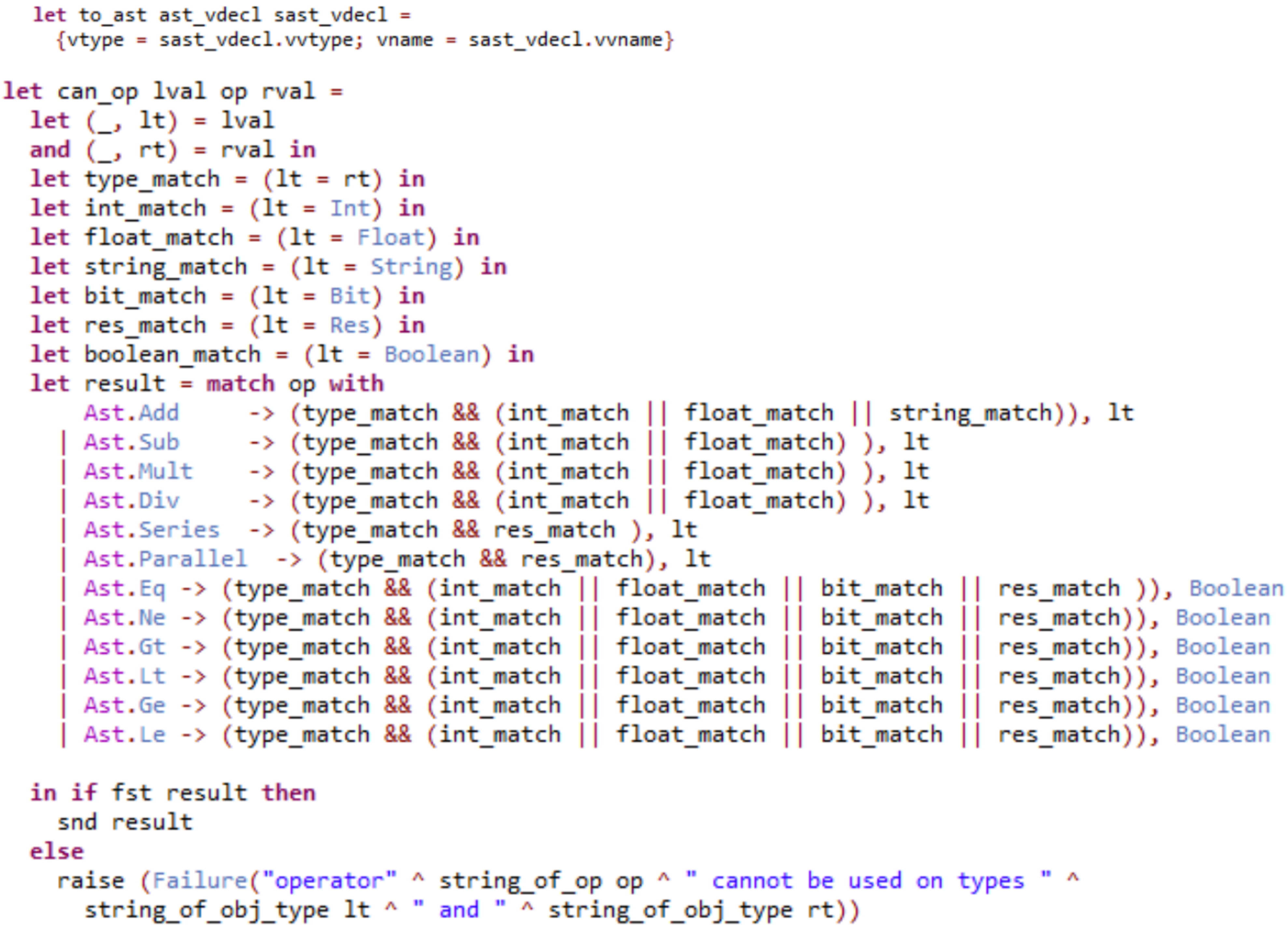
```
let rec find_function (scope : symbol_table) name = match name with
  "show" ->  {frtype = Void; ffname = "show";fformals=[];flocals=[];fbody=[] }
  | _ -> (try
    List.find (fun f -> f.ffname = name) scope.functions
  with Not_found ->
    match scope.parent with
    Some(parent) -> find_function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



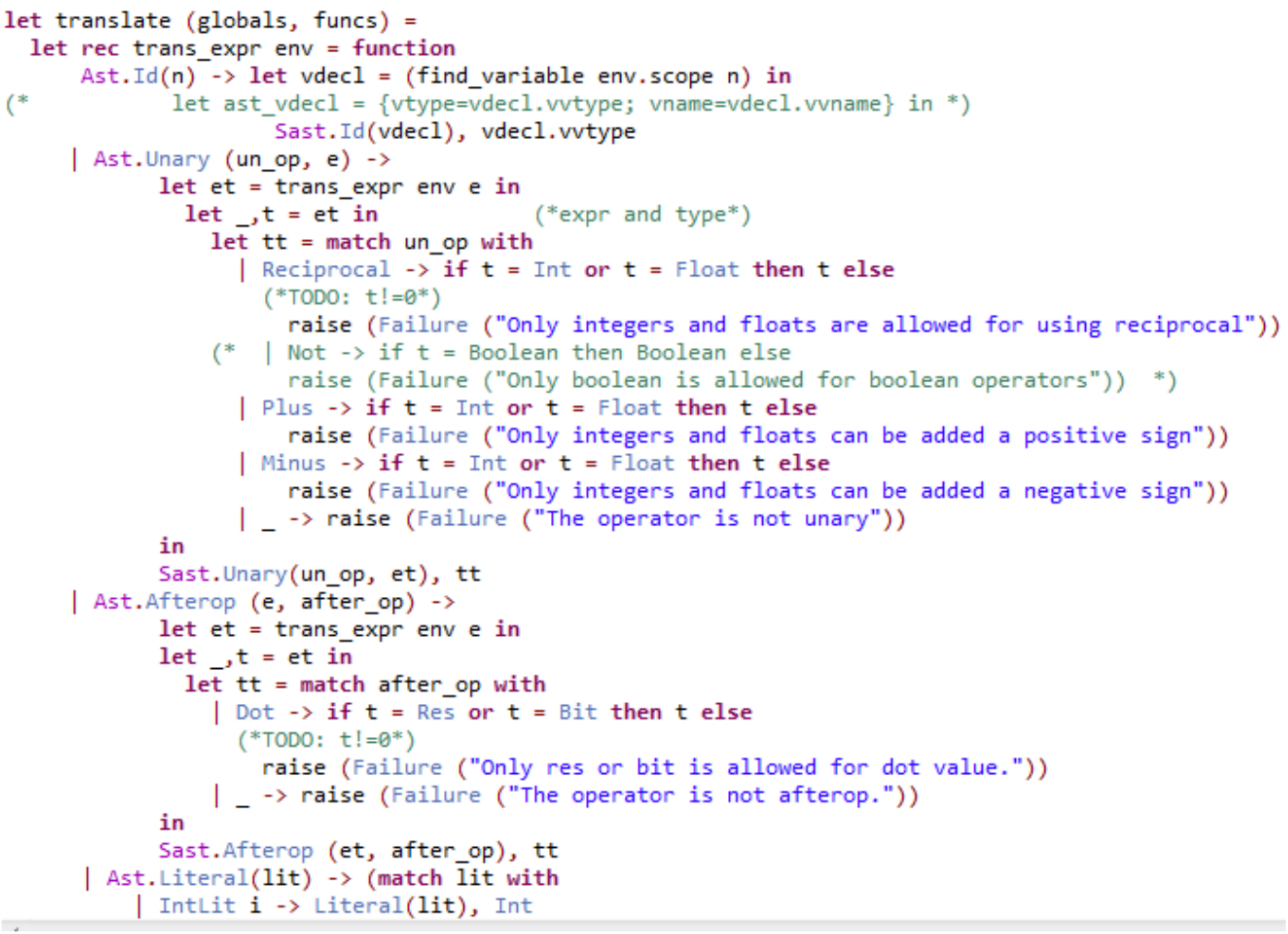
```
let rec find_function (scope : symbol_table) name = match name with
  "show" ->  {frtype = Void; ffname = "show";fformals=[];flocals=[];fbody=[] }
  | _ -> (try
    List.find (fun f -> f.ffname = name) scope.functions
  with Not_found ->
    match scope.parent with
    Some(parent) -> find_function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



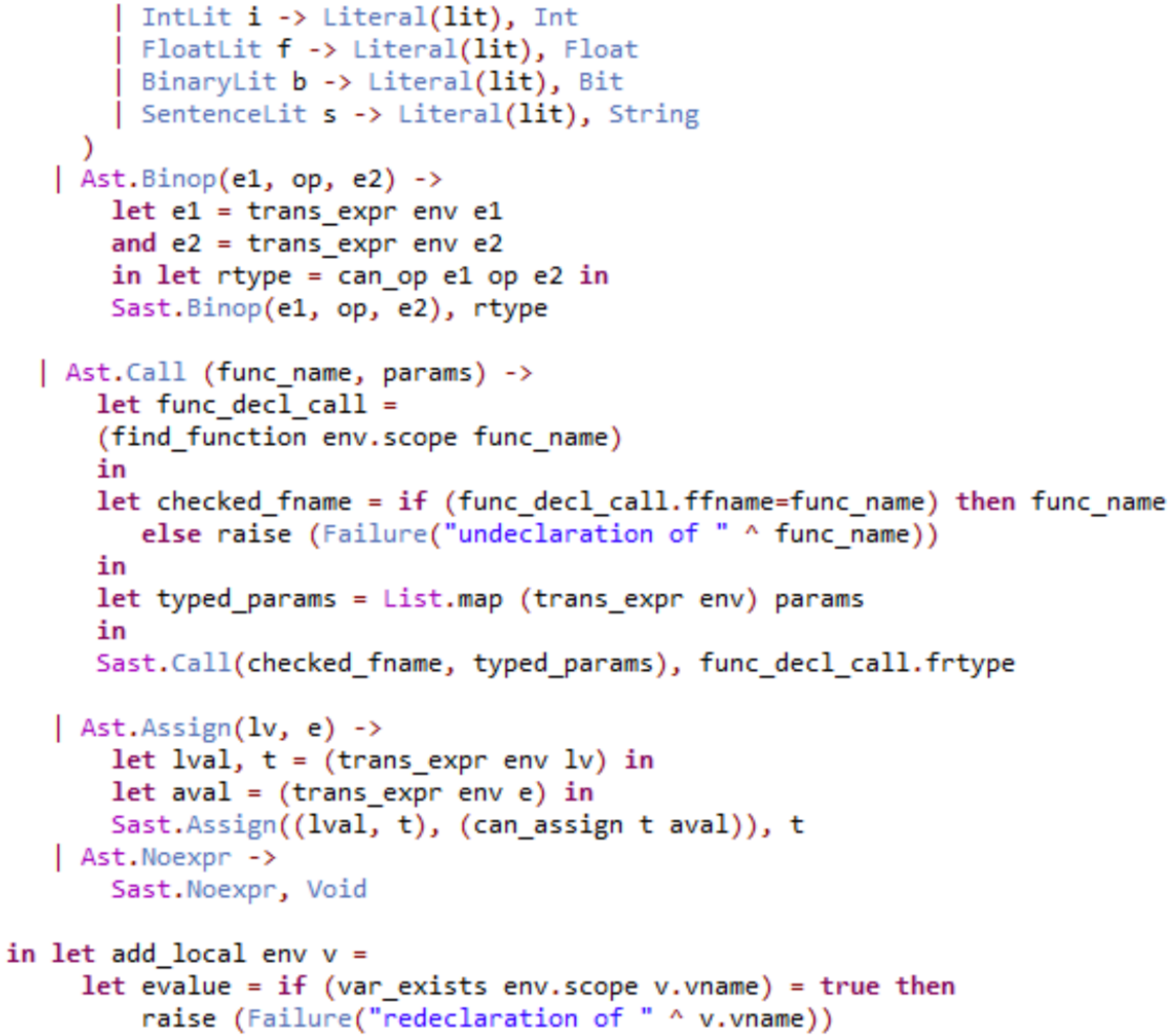
```
let can_op lval op rval =
  let (_, lt) = lval
  and (_, rt) = rval in
  let type_match = (lt = rt) in
  let int_match = (lt = Int) in
  let float_match = (lt = Float) in
  let string_match = (lt = String) in
  let bit_match = (lt = Bit) in
  let res_match = (lt = Res) in
  let boolean_match = (lt = Boolean) in
```



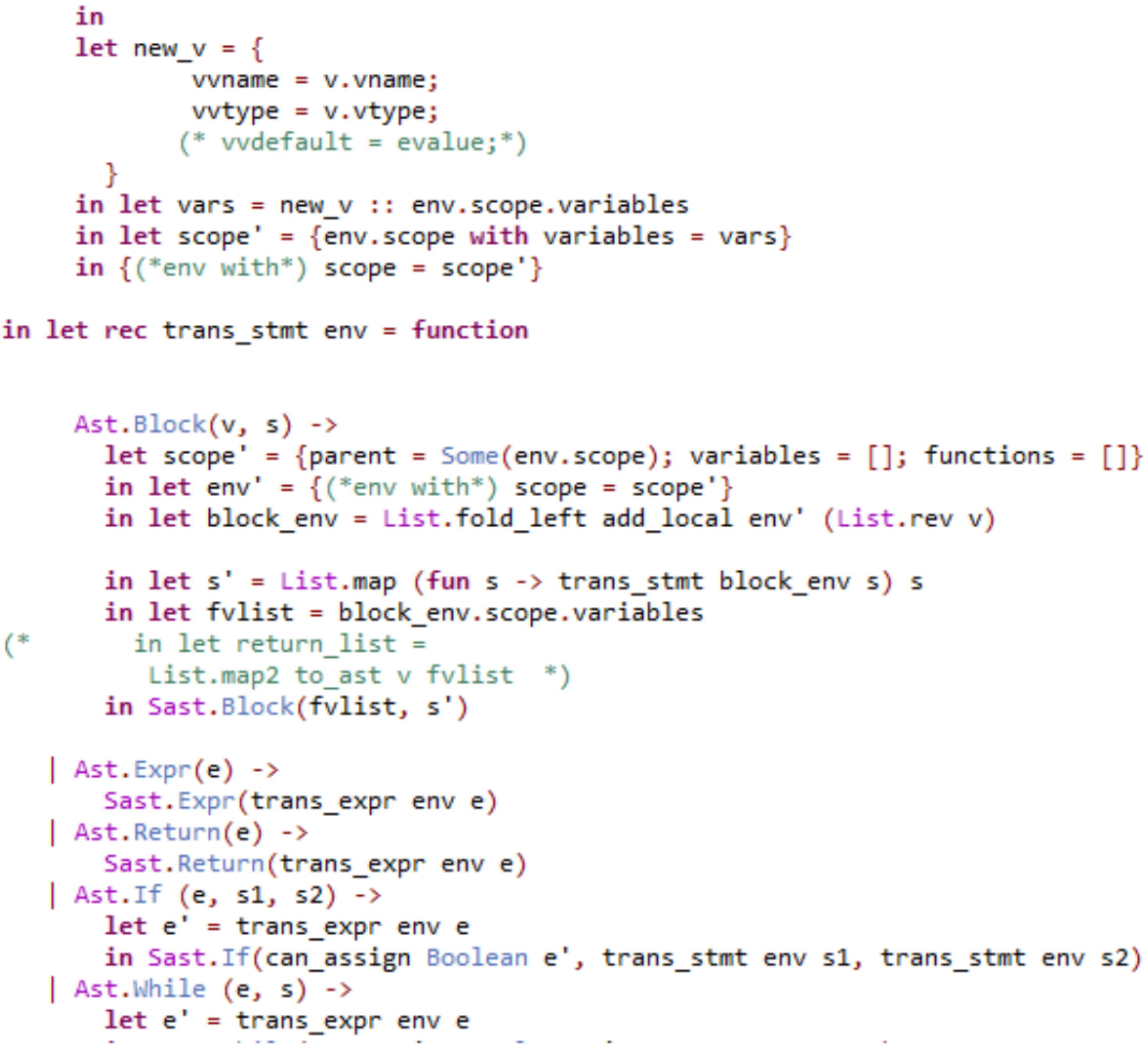
```
  let to_ast ast_vdecl sast_vdecl =
    {vtype = sast_vdecl.vvtype; vname = sast_vdecl.vvname}

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)), lt
    | Ast.Sub     -> (type_match && (int_match || float_match) ), lt
    | Ast.Mult    -> (type_match && (int_match || float_match) ), lt
    | Ast.Div     -> (type_match && (int_match || float_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
(*           let ast_vdecl = {vtype=vdecl.vvtype; vname=vdecl.vvname} 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
```

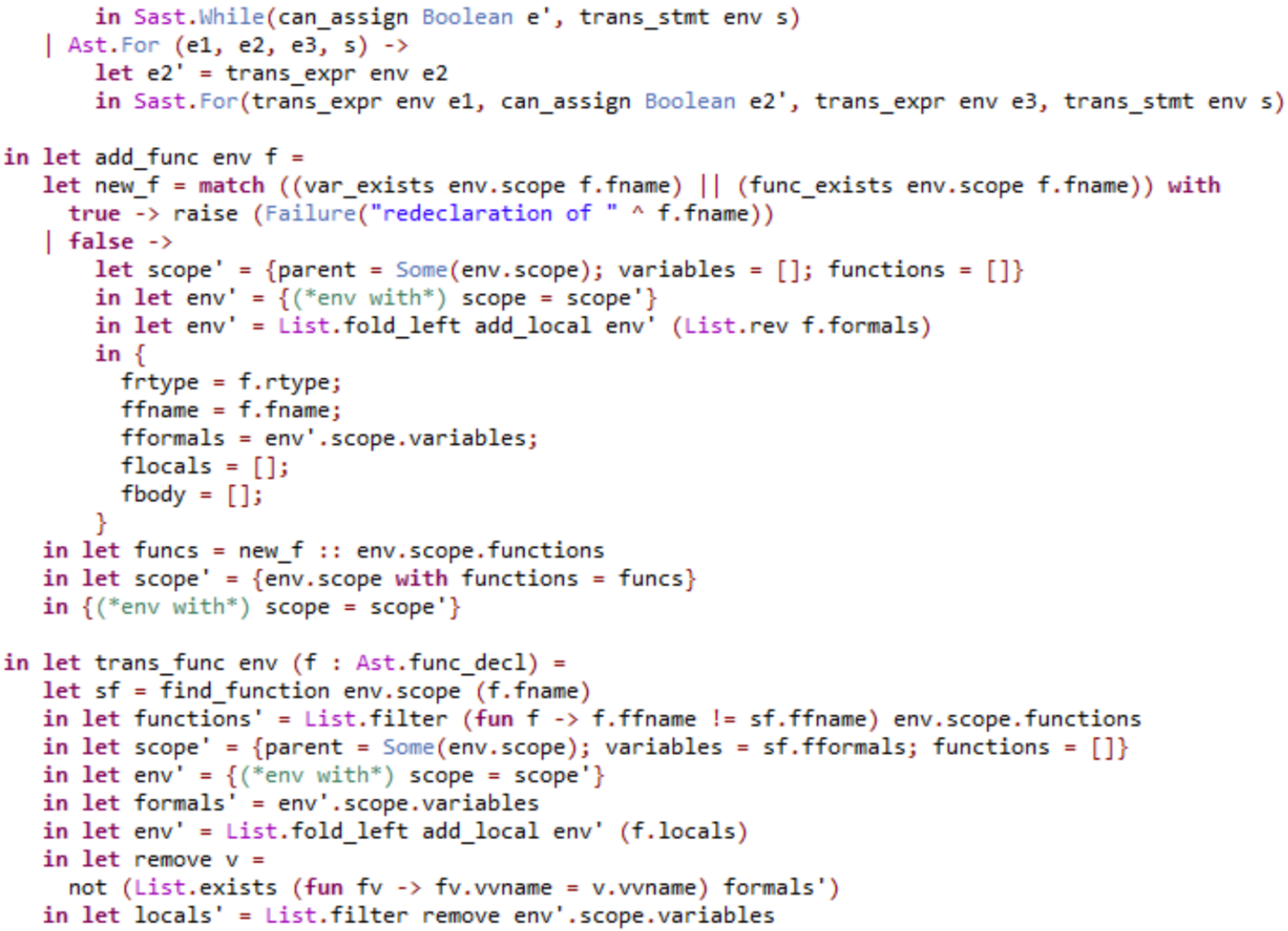


```
       | IntLit i -> Literal(lit), Int
       | FloatLit f -> Literal(lit), Float
       | BinaryLit b -> Literal(lit), Bit
       | SentenceLit s -> Literal(lit), String
     )
   | Ast.Binop(e1, op, e2) ->
       let e1 = trans_expr env e1
       and e2 = trans_expr env e2
       in let rtype = 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))
```

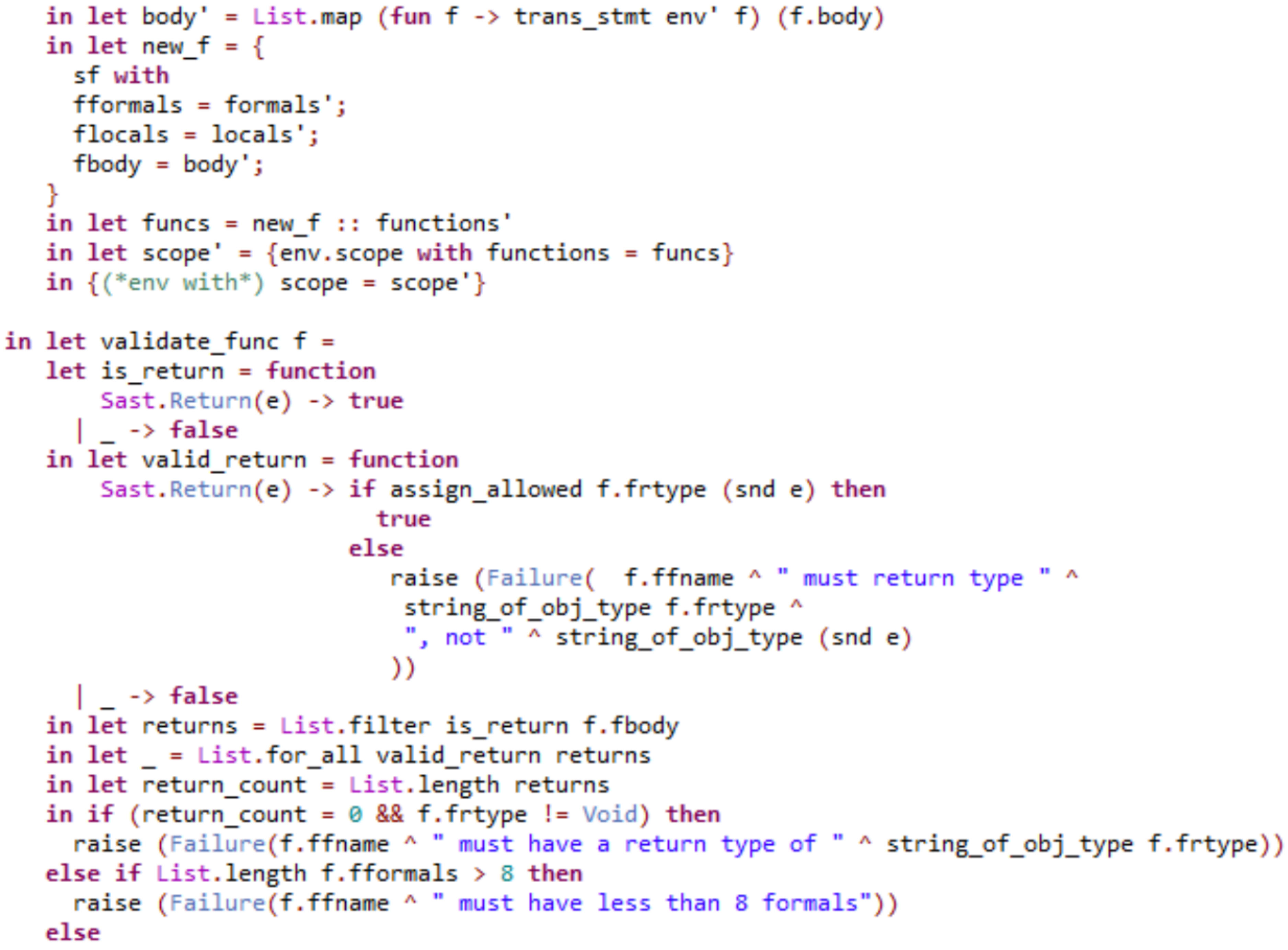


```
     in
     let new_v = {
             vvname = v.vname;
             vvtype = v.vtype;
            (* vvdefault = evalue;*)
       }
     in let vars = new_v :: env.scope.variables
     in let scope' = {env.scope with variables = vars}
     in {(*env with*) scope = scope'}

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 let return_list =
          List.map2 to_ast v fvlist  *)
       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)
```

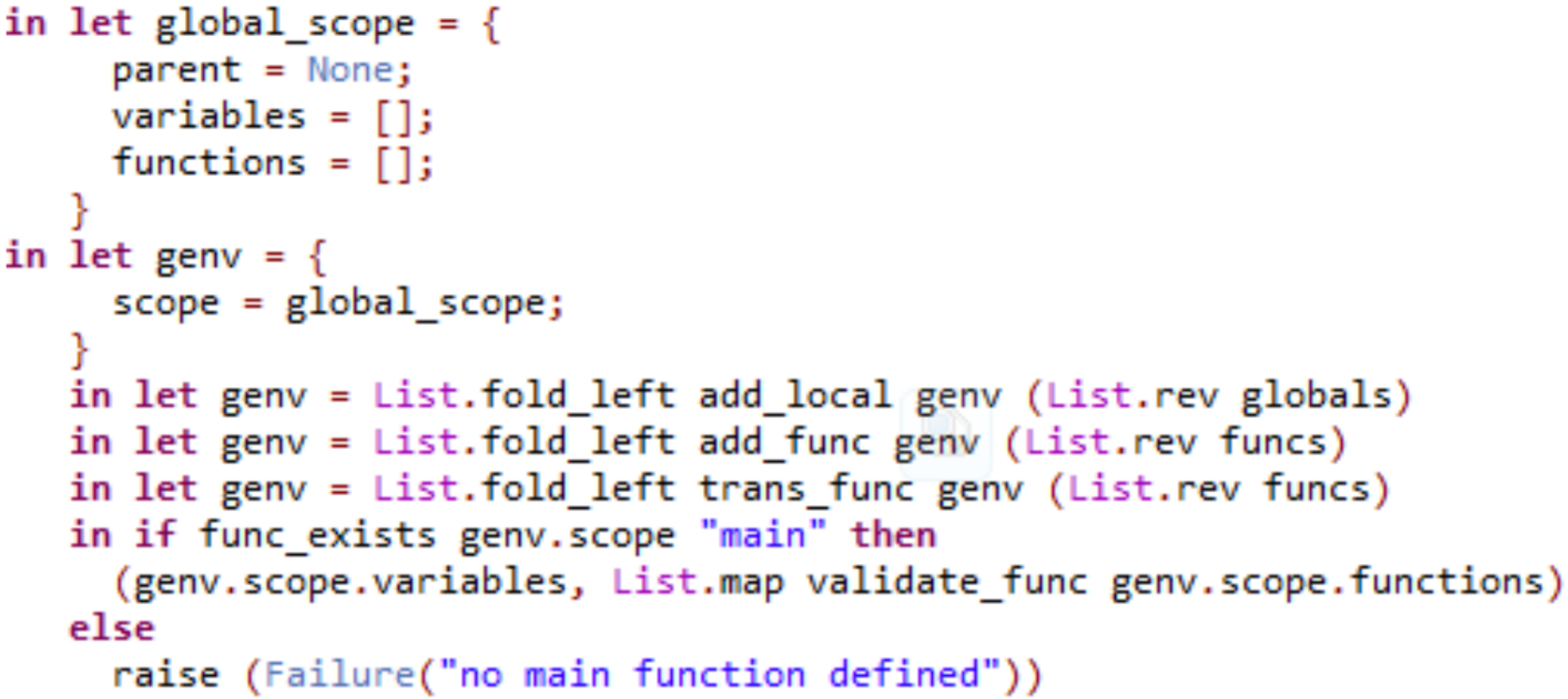


```
       in Sast.While(can_assign Boolean e', trans_stmt env s)
   | Ast.For (e1, e2, e3, s) ->
       let e2' = trans_expr env e2
       in Sast.For(trans_expr env e1, can_assign Boolean e2', trans_expr env e3, trans_stmt env s)

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
```



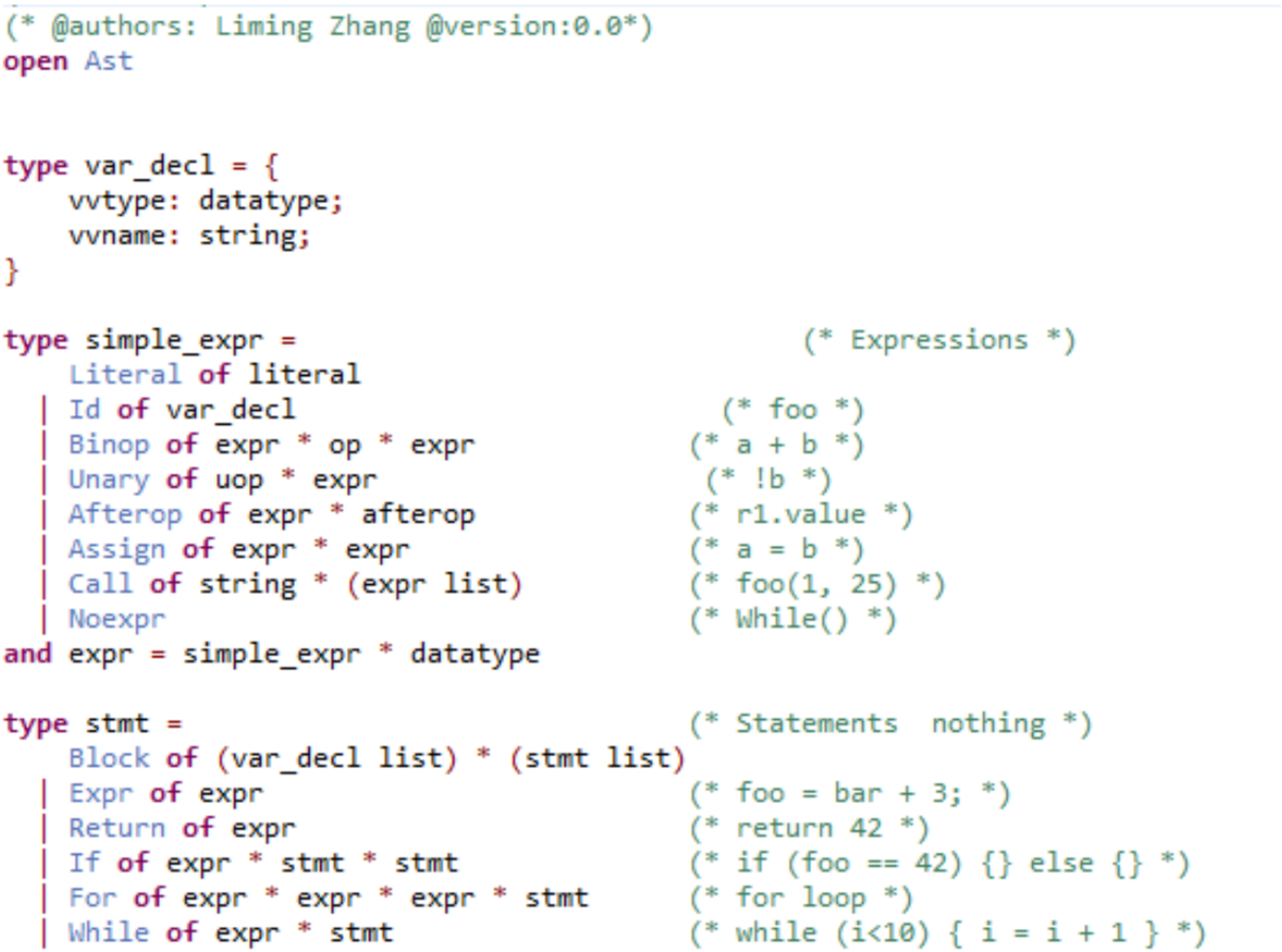
```
   in let body' = List.map (fun f -> trans_stmt env' f) (f.body)
   in let new_f = {
     sf with
     fformals = formals';
     flocals = locals';
     fbody = body';
   }
   in let funcs = new_f :: functions'
   in let scope' = {env.scope with functions = funcs}
   in {(*env with*) scope = scope'}

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

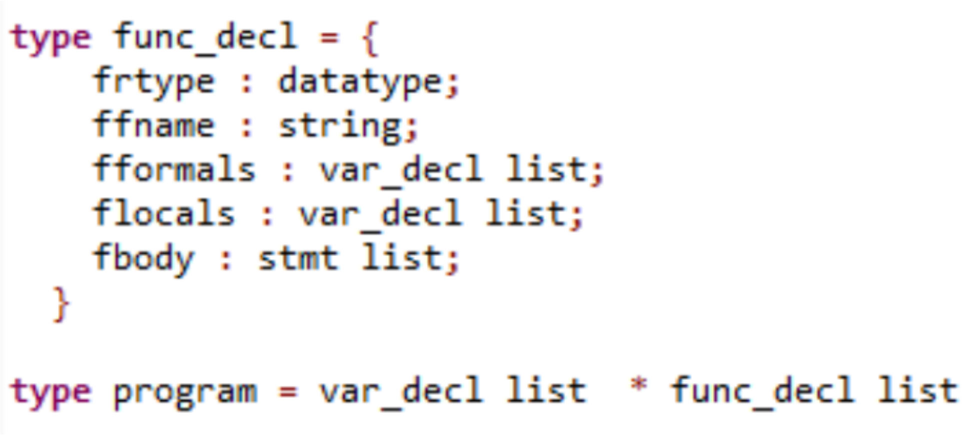


```
in let global_scope = {
     parent = None;
     variables = [];
     functions = [];
   }
in let genv = {
     scope = global_scope;
   }
   in let genv = List.fold_left add_local genv (List.rev globals)
   in let genv = List.fold_left add_func genv (List.rev funcs)
   in let genv = List.fold_left 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"))
```

SAST

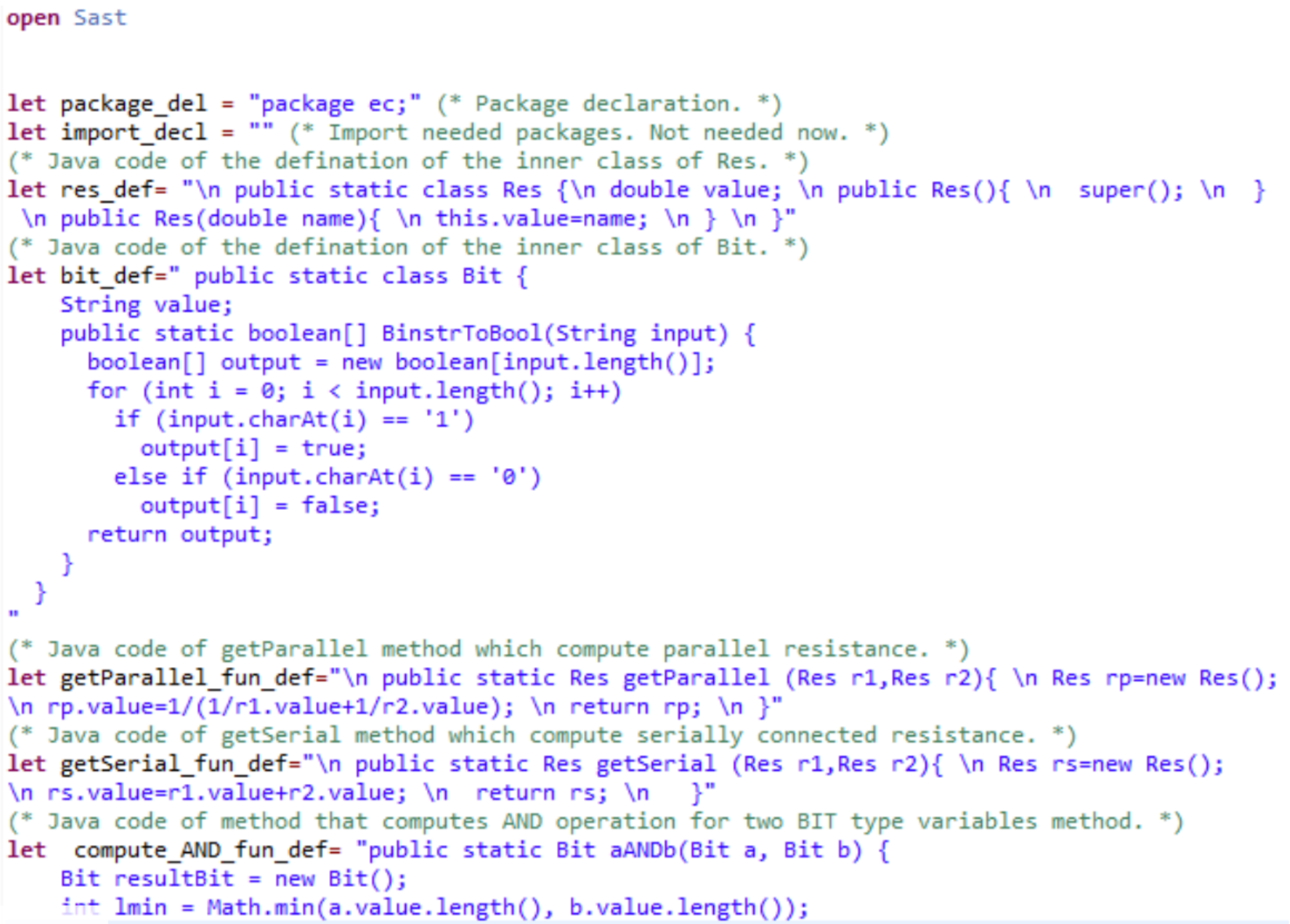


```
(* @authors: Liming Zhang @version:0.0*)
open Ast


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 } *)
```

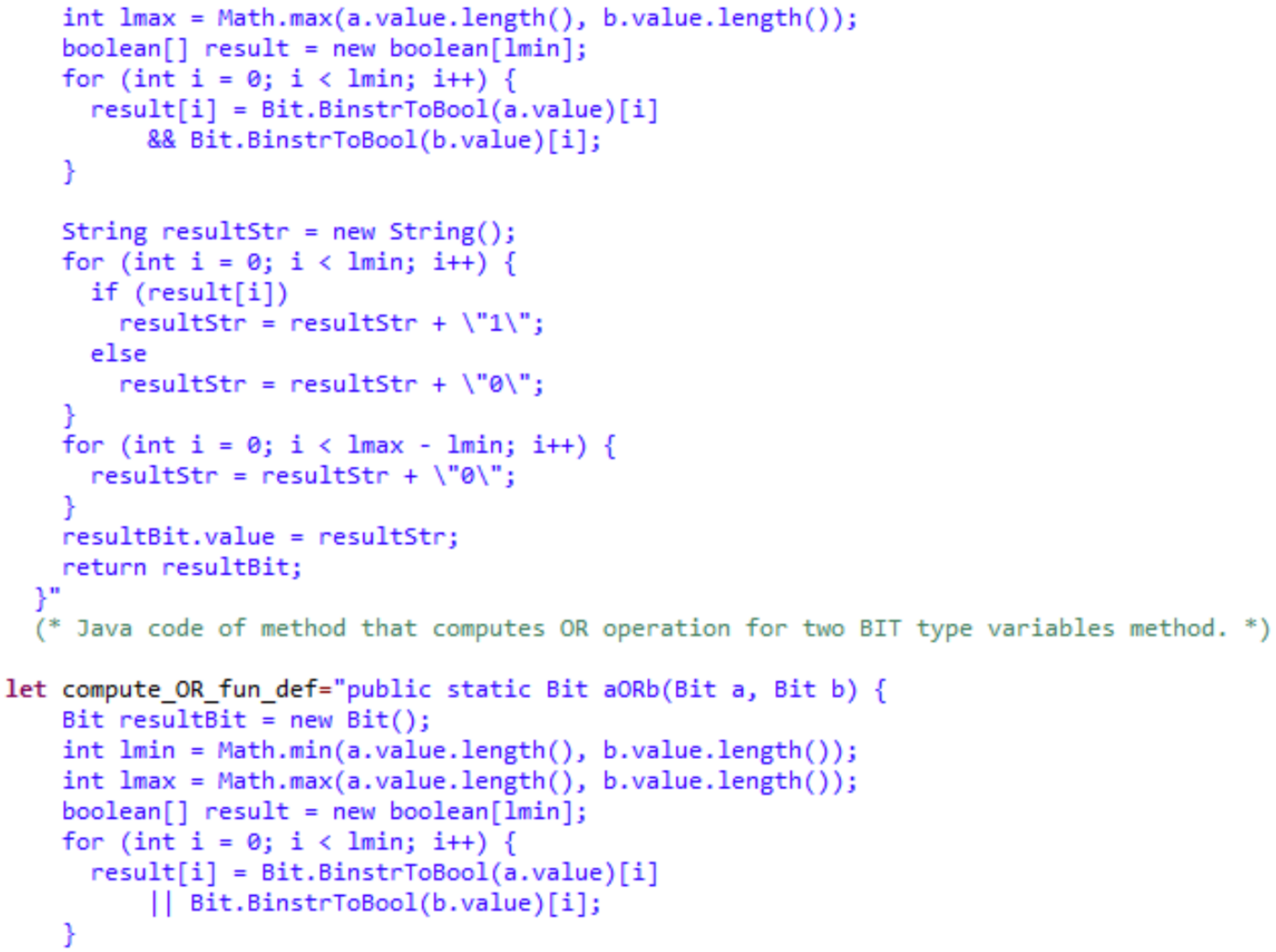


```
type func_decl = {
    frtype : datatype;
    ffname : string;
    fformals : var_decl list;
    flocals : var_decl list;
    fbody : stmt list;
  }

type program = var_decl list  * func_decl list
```

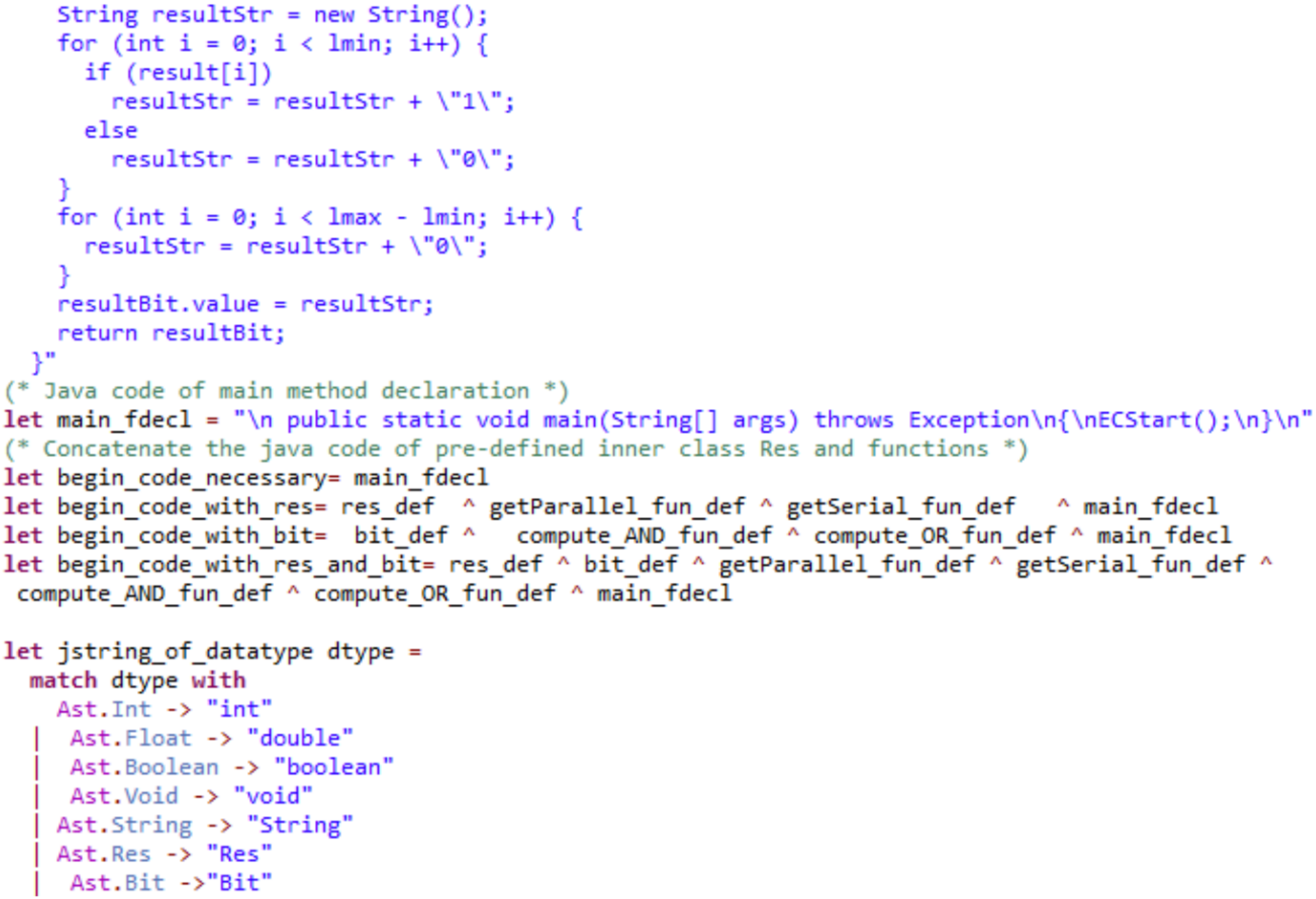
Translator



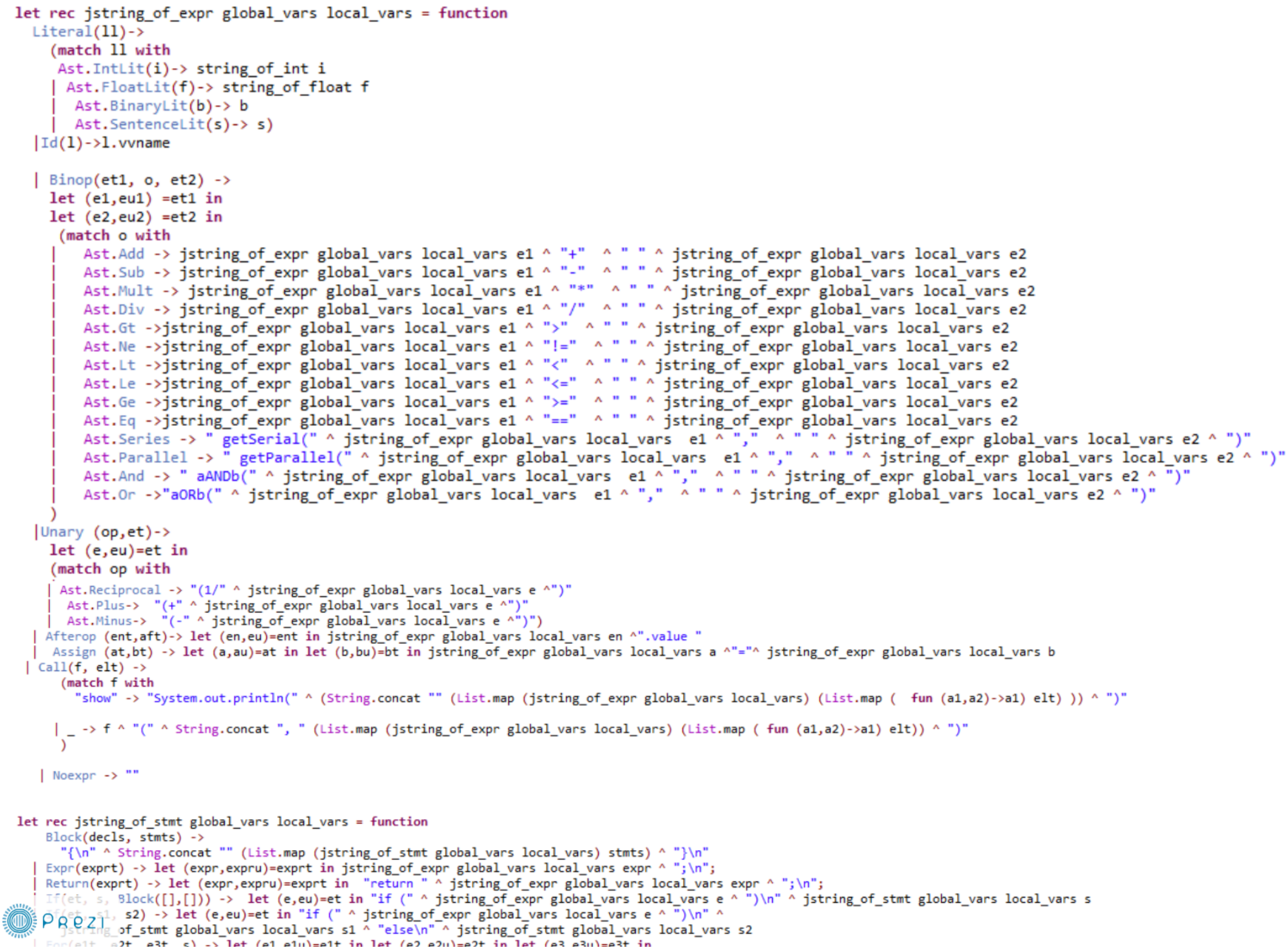
```
open Sast


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());
```



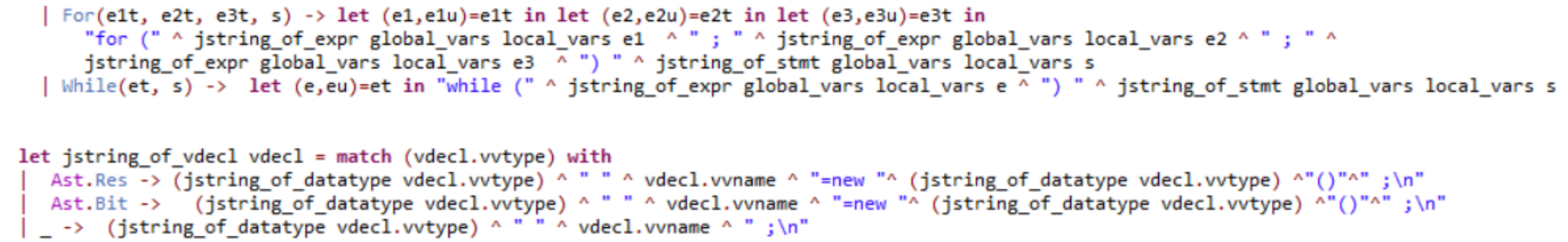
```
  int lmax = Math.max(a.value.length(), b.value.length());
  boolean[] result = new boolean[lmin];
  for (int i = 0; i < lmin; i++) {
   result[i] = Bit.BinstrToBool(a.value)[i]
     && Bit.BinstrToBool(b.value)[i];
  }

  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];
  }
```



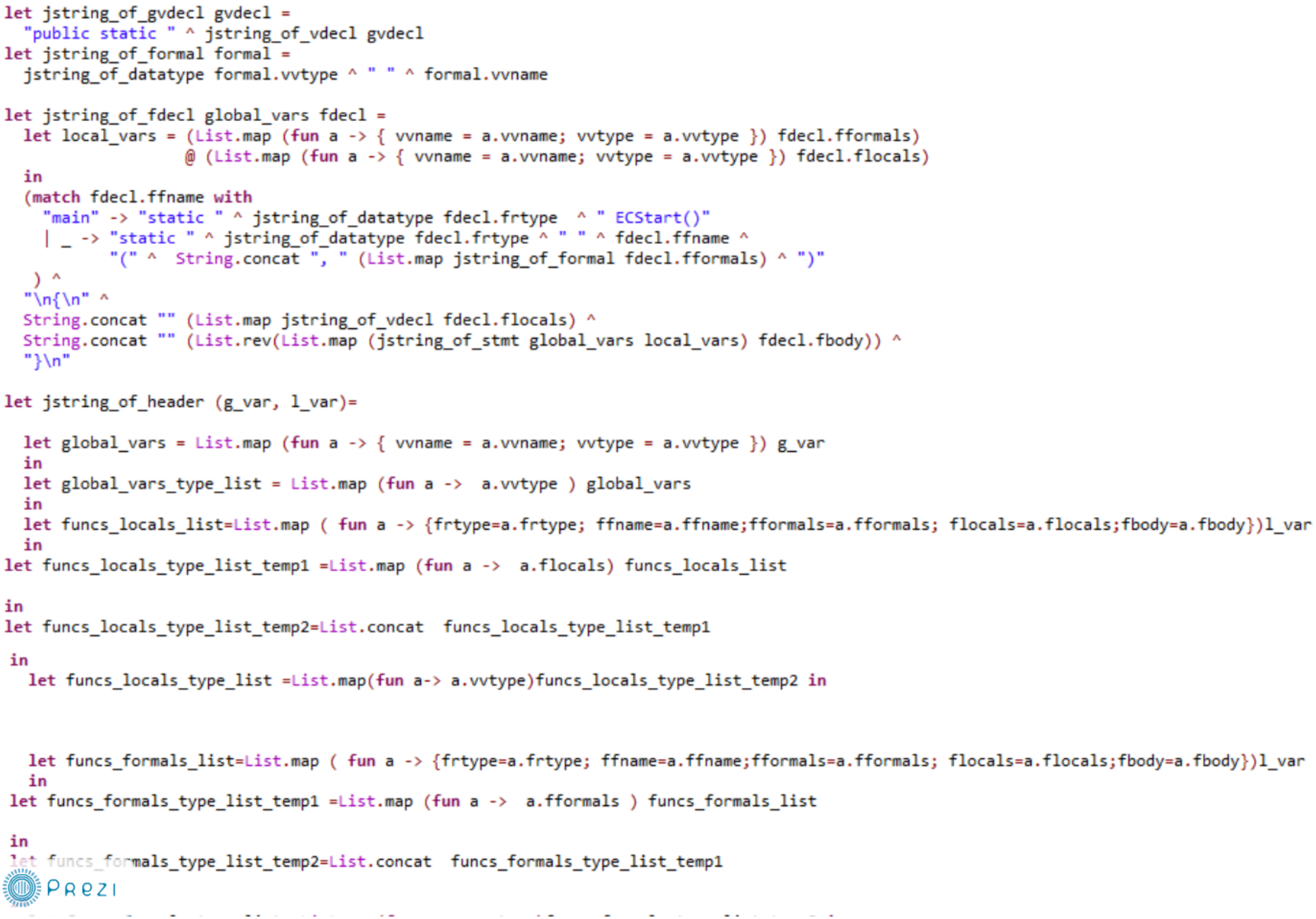
```
    String resultStr = new String();
    for (int i = 0; i < lmin; i++) {
      if (result[i])
        resultStr = resultStr + \"1\";
      else
        resultStr = resultStr + \"0\";
    }
    for (int i = 0; i < lmax - lmin; i++) {
      resultStr = resultStr + \"0\";
    }
    resultBit.value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

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"
```



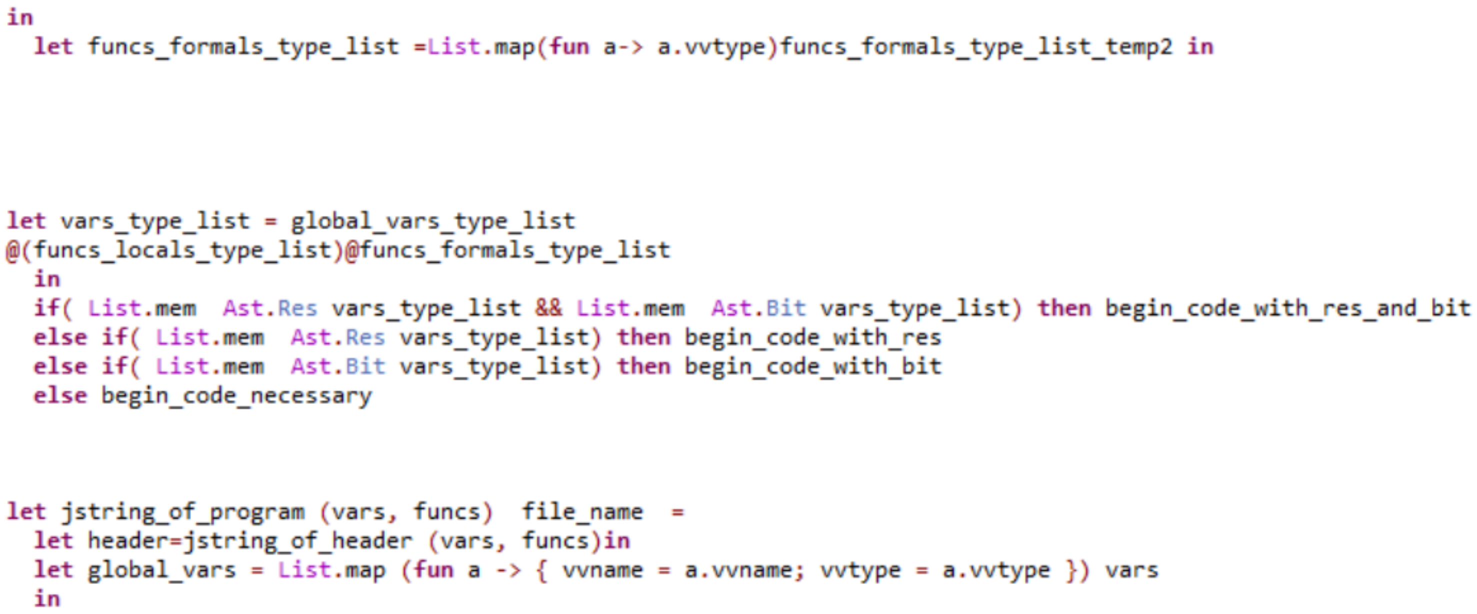
```
let rec jstring_of_expr global_vars local_vars = function
  Literal(ll)->
    (match ll with
     Ast.IntLit(i)-> string_of_int i
    | Ast.FloatLit(f)-> string_of_float f
    |  Ast.BinaryLit(b)-> b
    |  Ast.SentenceLit(s)-> s)
 |Id(l)->l.vvname

 | 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
```



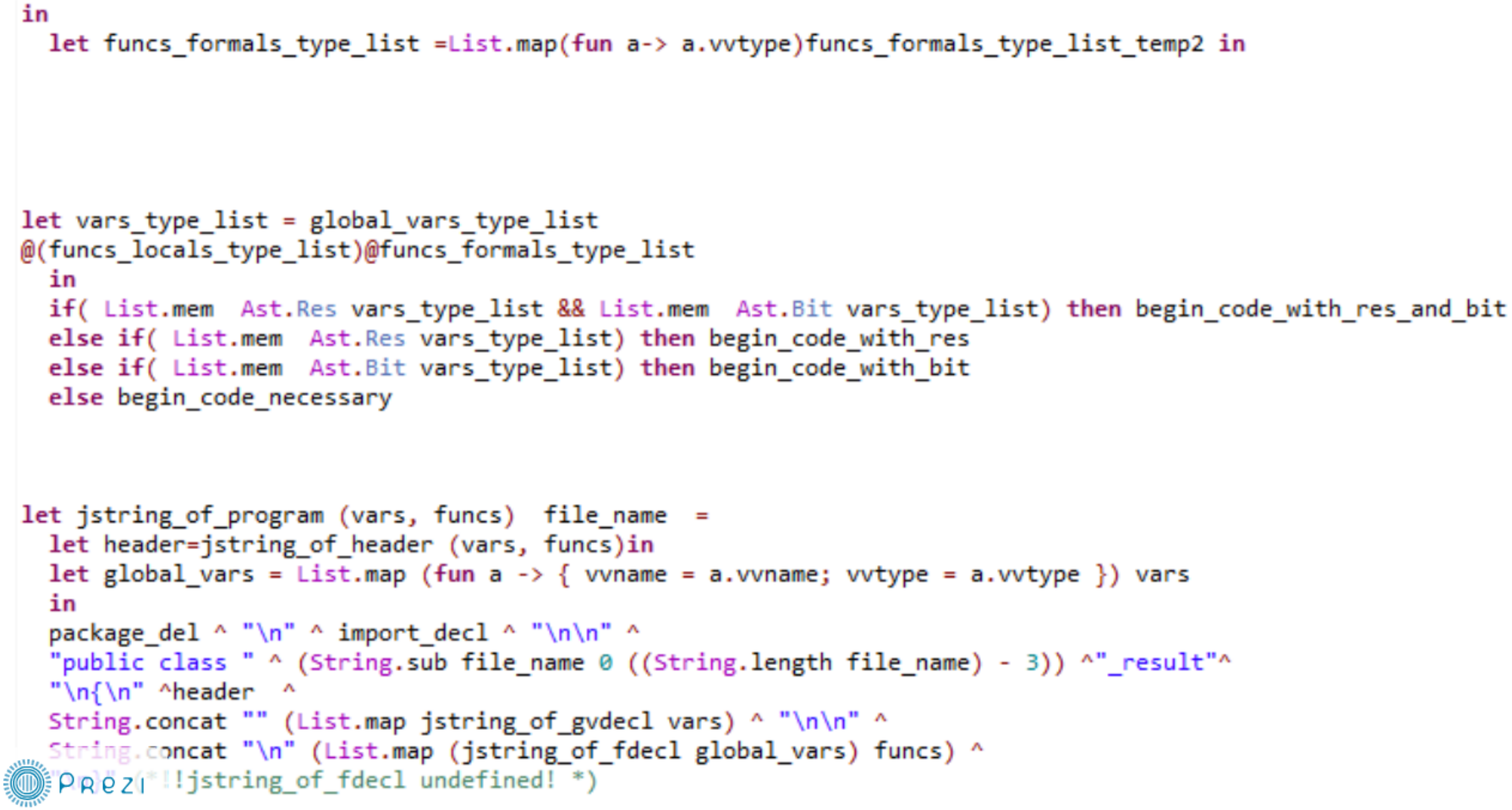
```
  | For(e1t, e2t, e3t, s) -> let (e1,e1u)=e1t in let (e2,e2u)=e2t in let (e3,e3u)=e3t in
     "for (" ^ jstring_of_expr global_vars local_vars e1  ^ " ; " ^ jstring_of_expr global_vars local_vars e2 ^ " ; " ^
     jstring_of_expr global_vars local_vars e3  ^ ") " ^ jstring_of_stmt global_vars local_vars 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
```



```
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}" (*!!jstring_of_fdecl undefined! *)
```

Lessons Learned

Early bird

Version control

Ocaml is strong type

Writing tests

Group work