CRYPS

CRYPTOGRAPHIC LANGUAGE

Hsiu-Yu Huang
hh2360@columbia.edu

Minita Shah
mjs2225@columbia.edu

Saket Goel
sg2679@columbia.edu

Sarfraz Nawaz
sn2355@columbia.edu

Table of Contents

1. INTRODUCTION

1.1 MOTIVATION

In today's world of technology and internet, wherein the primary mode of communication is via e-mails and many transactions are performed online, cryptography is of prime importance. It helps protect confidential information from being visible to everyone.

The motivation for Cryps comes from the fact that there is no language solely dedicated to Cryptography which has support for the large computations required for cryptographic algorithms. The idea was to create a language which has various operations on large numbers required in cryptography as built in operators and functions.

1.2. DESCRIPTION

CRYPS is a language designed to help regular users as well as programmers perform cryptographic operations. The language has support for generation and processing of large numbers in singular and collection forms , which is essential to implementation of common cryptography mechanisms and research on newer ones.

The language has features such as random number generation and prime number generation incorporated within it which are very essential in the implementation of cryptographic algorithms and thus enable users to create new algorithms easily. Such ready to use cryptography oriented features help save the user a lot of time which would otherwise be spent on defining these operations and also help reduce the size of the code considerably compared to other languages.

CRYPS is intended to be used by diverse backgrounds of people. Internet users can use it to encipher messages before exchanging confidential information over mail or instant messengers. Also, it could be used for E-commerce and banking transactions requiring web authentication. Enterprises with intranet systems can encrypt employee credentials with CRYPS. Most importantly, it can be used as a tool to develop new cryptographic algorithms.

2. LANGUAGE TUTORIAL

2.1. LANGUAGE TUTORIAL

In section 6, we present three implementations of standard Crytographic algorithms using Cryps. The examples demonstrate the usage of operators, functions and constructs in a Cryps program. For a more detailed explanation regarding the language syntax, please consult section 3 - the Language Reference Manual for Cryps.

2.2. PROGRAM FLOW

The Cryps programs could be either written to execute as standalone application or modules (as plugins) to be integrated with a Java application.
A standalone Cryps program would require a main() method defined which serves as the entry point for execution. Any function referred to in main() needs to be defined prior to its usage. There is no concept of function declarations.
On the other hand we could have programs written as separate modules that translate into a Java class file, which could be integrated with an external Java application. The modules developed here using Cryps would be specifically meant to implement a crypto function.

Cryps does not support global declaration of variables. The scope of a variable is the context within which it is defined, thus any variable declared within a function is only local to that function. Variables can be declared only within functions and passed to other functions.

The programmer can start a new scope any time when they create a statement block within { }, such as in iterative statements or if/else statements. However, functions cannot be defined within functions.

3. LANGUAGE MANUAL

3.1 LEXICAL CONVENTIONS

3.1.1 IDENTIFIERS

An identifier can consist of one or more letters, digits and underscore. The first character should be a letter followed by any sequence of letters, digits and underscore character. First ten characters are significant. Uppercase and lowercase letters are different.

3.1.2 COMMENTS

Comments start with '(#' and are terminated with '#)'. A common convention is followed for both single line and multiple line comments.

3.1.3 WHITESPACE

The only way to represent whitespace is by binding it with double quotes (" ") on either side, all other forms of whitespace are not considered.

3.1.4 RESERVED KEYWORDS

int string print eof

if	else	for	while
return	~mod		

3.1.5 DATA TYPES

The fundamental data type is **int.** Apart from the fundamental data type, there is a class of derived types constructed from fundamental data types in the following ways:

1. Arrays of objects of **int** - The syntax for an array is an identifier followed by '[' and ']'.
2. Matrix of objects of **int** – The syntax for a matrix is an identifier followed by '[' and ']' '[' and ']'.
3. Functions which return objects of the type int.

3.1.6 CONSTANTS

CRYPS has only integer constants.

3.1.7 SEPARATORS

The ',' symbol is used to represent a comma separated list. The symbol ';' is used to indicate the end of a statement.

3.1.8 SCOPE RULES

CRYPS uses static scoping. It is a block structured language, and the scope of names declared in a block is within the body of the block.

3.2 EXPRESSIONS AND OPERATORS

Symbol	Operations	Associativity
+	Addition	Left associative
-	Subtraction	Left associative
*	Multiplication	Left associative
/	Division	Left associative
^	Exponential	Left associative
%	Modulus	Left associative
=	Equal to	Left associative
!=	Not equal to	Left associative
<	Less than	Left associative
<=	Less than or Equal to	Left associative
>	Greater than	Left associative
>=	Greater than or Equal to	Left associative
<<	Left Shift	Right associative
>>	Right Shift	Right associative
@	EXOR	Left associative
,	Sequence	Left associative
&&	Logical AND	Left associative
||	Logical OR	Left associative
<-	Assignment	Right associative
~mod	Modulus Inverse	Left associative
<>	GCD	Left associative
!	Not	Left associative

3.3 CONTROL STRUCTURES

Conditional Statement

if (expression) statement

if (expression) statement else statement

While statement
    while (expression) statement

For statement
    for (expression ; expression ;expression) statement

3.4 NAMESPACE

CRYPS has only one name space.

3.5 GRAMMAR

program -> block

block -> stmt_list

stmt_list -> ε | stmt+

stmt -> expr';' | '{' block '}' | while-stmt | if-stmt | for-stmt | return-stmt';' | variable-initialization-list';' | variable-assign';' | function-defn | print-stmt';'

function-defn -> data-type id '(' formal-arg-val ')' stmt-list

formal-arg-val -> ε | formal-arg-list

formal-arg-list -> formal-arg | formal-arg-list ',' formal-arg

formal-arg -> data-type id

data-type -> int

variable-initialization-list -> var-init | var-init-list ',' var-init

var-init -> id | id assign expr | id '['literal']' | id '['literal']''['literal']' | id'['']' assign digit-set
    | id'['']''['literal']' assign digit-mat-set

digit-set -> ‘{‘digit-list’}’

digit-list -> literal | digit-list’,’literal

digit-mat-set -> ‘{‘digit-mat-list’}’

digit-mat-list -> digit-list | digit-mat-list’;’digit-list

variable-assign -> id’[‘’]’ assign digit-set | id ‘[‘’]’’[‘literal’]’ assign digit-mat-set

print-stmt -> print-val-list’;’

print-val-list -> print-val | print-val-list ’+’ print-val

print-val -> id | P_str

if-stmt -> if expr stmt | if expr stmt else stmt

while-stmt -> while expr stmt

for-stmt -> for expr’;’expr’;’expr stmt

expr -> expr binop expr | unary expr | ‘{‘expr’}’ | assign expr | literal | function-call |
array-val | matrix-val | array-assign-expr | matrix-assign-expr

binop -> '+' | '-' | '*' | '/' | '^' | '%' | '<' | '>' | '>=' | '<<' | '>>' | '@' | ‘&&' | ' || ' | '!=' | '=' |
‘<>’

assign -> ‘<-‘

unary -> ‘!’ | ‘~mod’

array-val -> id '[' expr ']'

matrix-val -> id '[' expr ']' '[' expr ']'

array-assign-expr -> id '[' expr ']' assign expr

matrix-assign-expr -> id '[' expr ']' '[' expr '] assign expr

function-call -> id ‘(‘ actual-arg-list ’)’

actual-arg-list -> ε | actual-arg

actual-arg -> expr | actual-arg’,’expr

literal -> ['0' - '9']

id -> ['a'-'z' 'A'-'Z'] ['a'-'z' 'A'-'Z' '0'-'9' '_']*

return-stmt -> return | return expn | return id | return literal

4. PROJECT PLAN

4.1 PROCESS USED

4.1.1. PLANNING AND SPECIFICATION

Our group was excited about implementing a cryptographic language. We came up with several ideas involving string manipulation and digital signatures but our initial meetings before submitting the proposal did not bring clarity as to how we would go about implementing it. Professor Edwards and our project in charge Rajesh were instrumental in giving us a direction on how to proceed. The specifications for Cryps were laid out for the first time as we were preparing the Language Reference Manual. Our aim was to include basic features common to all languages and few but powerful features aimed to help develop a cryptographic algorithm easily. Post submission of the Language Reference Manual we met twice, almost every week. We set targets for the following week and divided the work among the team members with the focus that we need to cover all the basic features of the language first and then add the special features. We occasionally worked in pairs so that we could help each other when we were in doubt. Our language specifications changed over time with one of the most important being focusing on biginteger operations rather than working with many data types.

4.2.2. DEVELOPMENT AND TESTING

Each member of the group, with help from other members in case of doubts, wrote and tested his code. The weekly meetings focused on integrating the code and explaining the functionality to one another. The entire project development process involved active participation of all the team members with us working in pairs and trying to work together in the same room as a group as much as possible. We made steady, significant progress each

week and many important concepts which we had not understood earlier became clear as we progressed. We used Google code as our code repository Rapid SVN for version control.

4.2 PROGRAMMING STYLE GUIDE

Some of the rules we set for ourselves:

1. We worked simultaneously on the lexer, parser and ast files starting with a basic set of binary operations. Thereafter, incremental adding of features by an individual included adding the corresponding code in all the three files.
2. After every meeting we followed up with the minutes of the meeting. It helped us gauge the progress made in each meeting and also the amount of work done.
3. We tried to make every variable name meaningful and self-explanatory.
4. Our mentor was sent gross weekly updates so that he could keep track of our progress.
5. We faced problems of debugging in OCAML so we followed the approach of debugging small modules at a time and then adding to them.

4.3 PROJECT TIMELINE

09/15/08	Project group formed
09/18/08	Language decided
09/24/08	Language proposal submitted
09/22/08	Language Reference Manual submitted
09/30/08	Lexer completed, Parser worked on
11/02/08	Compiler (printer.ml) started
11/25/08	Parser completed, began developing test cases
12/18/08	Compiler (printer.ml) and Testing completed
12/19/08	Final presentation
12/19/08	Final Report

4.4 ROLES AND RESPONSIBILITIES

Compiler, Algorithm in Cryps, Rapid SVN, Makefile, Project Report, Interpreter(now excluded)	Hsiu-Yu Huang
Lexer, Parser, Test cases, Algorithm in CRYPS, Project Report	Minita Shah
Compiler, Test cases, Algorithm in CRYPS, Project Report, Grammar, Interpreter(now excluded)	Saket Goel
Lexer, Parser, Type Checking, Scoping, Project Report	Sarfraz Nawaz

4.5 SOFTWARE DEVELOPMENT ENVIRONMENT TOOLS

Programming

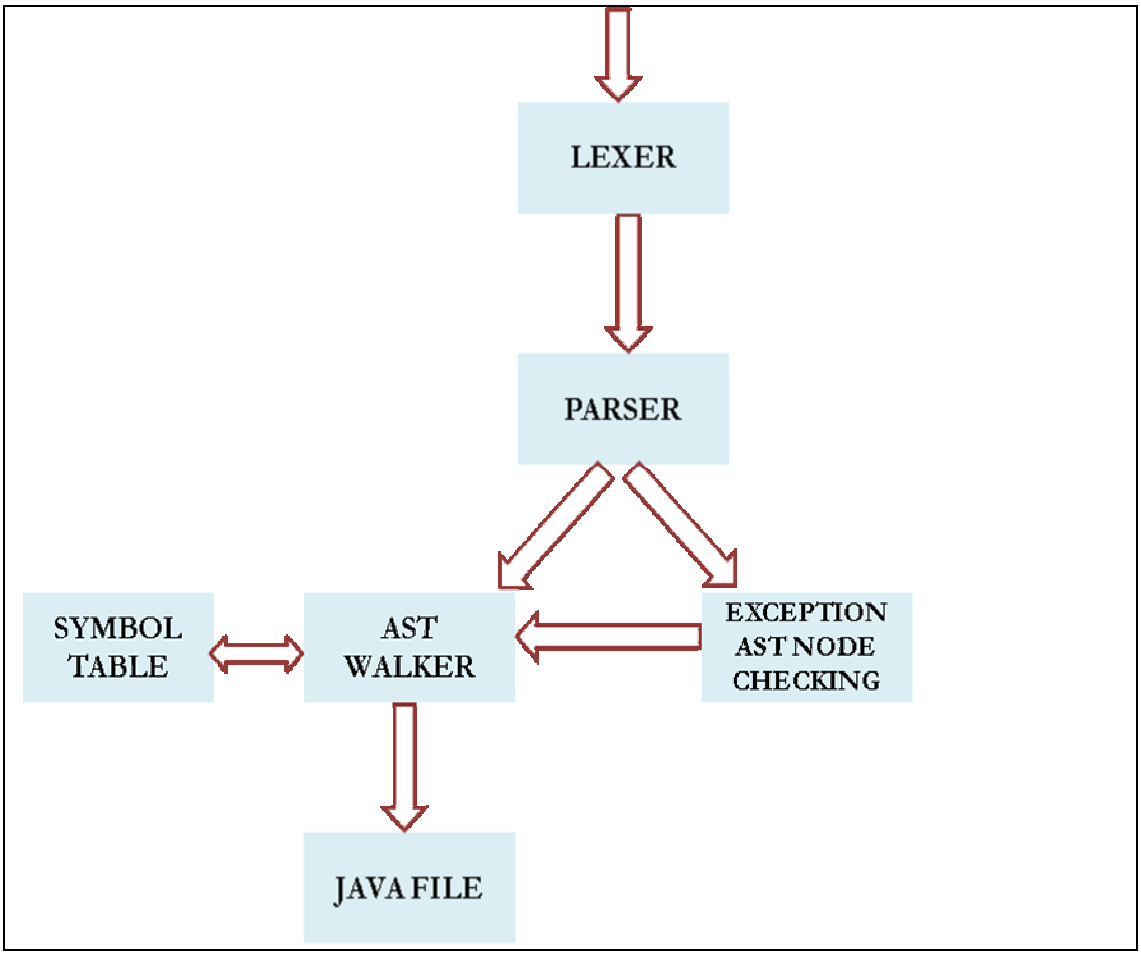
- Ocaml
- Java

Tools

- msys shell for Ocaml.
- Eclipse for Java.
- Google Code and Rapid SVN for source code repository and version control.
- Ocaml Lex and Yacc tools for lexer and parser.
- Ocaml Dep for checking dependency rules.
- Windows and Linux environments.

5. ARCHITECTURAL DESIGN

5.1 BLOCK DIAGRAM OF CRYPS TRANSLATOR



Flow Chart: Depicts information flow between the components of the translator

5.2 DESCRIPTION OF ARCHITECTURE

The Cryps source file consists of a program possibly consisting of constructs, operators and other statements written in Cryps language. This file is given as input to the Lexer.

The Lexical Analyzer consists of tokens defined for operators, identifiers, keywords and others such as comments. The role of the Lexer is to take the program written in Cryps as input and convert it into an equivalent stream of tokens thereby removing any whitespaces present. This sequence of tokens is then passes on as input to the Parser.

The Parser specifies the precedence and associativity of all the operators and also has the syntactic rules defined within it. Primarily, the Parser defines the set of grammatical rules to be followed for a program written in Cryps language. The Parser thus takes the stream of input tokens and builds an abstract syntax tree implicit in the input tokens.

The abstract syntax tree is basically a tree representation of the simplified syntactic structure of the source code in Cryps and follows the structure of having the leaf nodes represent the operands and their parent nodes represent the constructs and operators.

The AST Walker is the printer.ml file which on being receiving the input from the Parser perform AST node checking as well as checks for scoping issues. These checks are performed by referring to the details stored in the Symbol Table.

The Symbol Table is a hash table which consists of information associated with identifiers in the program such as it's type.

At every operator encountered in the Abstract Syntax Tree, a check is performed to see if the types of the operands are the same as expected. The printer.ml file then generates a Java file which consists of Java code equivalent to the Cryps source code program.

6. SAMPLE PROGRAMS

Below are three cryptographic algorithms implemented in Cryps:

(# Affine Transformation Algorithm #)

```
(# Function to encrypt #)
int encrypt(int mess, int key1, int key2)
{
        int cipher, mod1 <- 26;
        cipher <- (key1 * mess + key2) % mod1;
        return cipher;
}

(# Function to encrypt #)
int decrypt(int cipher, int key1, int key2)
```

```
{
        int plaintext, mod2 <- 26;
        plaintext <- ((key1 ~mod mod2) * (cipher - key2)) % mod2 ;
        return plaintext;
}


(# main function - entry point to the program #)
int main()
{
        int key1 <- 7, key2 <- 3;
        int message <- 7, modulus <- 26;
        int cipher, plaintext;

        if ((key1 <> modulus ) = 1)
        {
                cipher<-encrypt(message, key1, key2);
                print("Encrypted output: " + cipher);

                plaintext<-decrypt(cipher, key1, key2);
                print("Decrypted output:" + plaintext);
        }
}
```

(# Transposition Algorithm #)

(# Function conducts encryption and decryption by transposing the matrix populated with digits of a number #)

```
int transposition(int num, int numDigits)
{
        int cnt <- 0, i , j, exponent, d;
        int n <- 4;
        int c[500][500],f[500][500],a[500];
        int result, digCount<-numDigits-1;

        for(i<-temp; i>=0; i<-i-1)
        {
                exponent <- 10^i;
                d <- num/exponent;
                a[digCount-i] <- d;
                num <- num%exponent;
        }

        for (i<-0;i<n;i<-i+1)
        {
                for(j<-0;j<n;j<-j+1)
                {
                        if(cnt<numDigits)
                        {
```

```
                c[i][j] <- a[cnt];
                cnt     <-cnt+1;
            }
            else
            {
                c[i][j] <- 0;
            }
        }
}

(# Function to encode #)
for (i<-0;i<n;i<-i+1)
{
        for(j<-0;j<n;j<-j+1)
        {
            f[i][j] <- c[j][i];
        }
}

print("The encoded key is : \n");
for (i<-0;i<n;i<-i+1)
{
        for(j<-0;j<n;j<-j+1)
        {
            result <- f[i][j];
```

```
                print(result);
        }
    }

    (# Function to Decode #)
    for (i<-0;i<n;i<-i+1)
    {
        for(j<-0;j<n;j<-j+1)
        {
            c[i][j] <- f[j][i];
        }
    }

    print("The decoded key is :\n");
    for (i<-0;i<n;i<-i+1)
    {
        for(j<-0;j<n;j<-j+1)
        {
            result <- c[i][j];
            print(result);
        }
    }
    return 0;
}
```

```
(# Main function – entry point to the program #)
int main()
{
        int number <- 12345678, length <- 8;
        print("The key to be encoded is : "+number);
        transposition(number ,length);
}
```

(# RSA algorithm implementation #)

```
(# Encyption function #)
int encrypt(int C, int e, int M, int n)
{
        int j;
        C <- 1;
        for(j<-0;j<e;j<-j+1)
        {
                C <- C * M % n;
                C <- C % n;
        }
        print("\n\tEncrypted keyword : " + C);
```

```
        return C;
}

(# Decryption function #)
int decrypt(int C,int d,int M,int n)
{
        int k;
        M <- 1;
        for(k <- 0;k < d;k<-k+1)
        {
                M <- M*C%n;
                M <- M%n;
        }
        print("\n\tDecrypted keyword : %" + M);
        return M;
}

(# Function to check if there is a number relatively prime to 2 input numbers #)
int check(int e,int phi,int flag)
{
        int i<-3;
        int fst <- e%i, snd <- phi%i;
        for(i <- 3; (fst = 0 && snd = 0);i <- i+2)
        {
                flag <- 1;
```

```
            return flag;
        }
            flag <- 0;
            return flag;
        }
}

int main()
{
        int p<-7,q<-17,s;
        int phi, M, n, e, d, C, flag<-1;

        n <- p*q;
        phi<-(p-1)*(q-1);
        print("\n\tF(n)\t<- " + phi);

        while(flag=1)
        {
            flag<-0;
            e<-5;
            flag<-check(e,phi, flag);
        }
        d <- 1;

        while(s!=1)
```

```
        {
                s <- (d*e)%phi;
                d<-d+1;
        }
        d <- d-1;

        print("\n\tPublic Key\t: " + e + n);
        print("\n\tPrivate Key\t:"+ d + n);

        M <- 19;
        encrypt(C,e,M,n);

        C <- 66;
        decrypt(C,d,M,n);
}
```

7. TEST PLAN

Sample test cases are as follows:

```
(*Test Case 1*)

int main()
{
```

```
int a <- 100;
int b <- 25;
int c;
c <-a + b;
print(c);
c<-a+b;
        print(c);
c<-10+16;
        print(c);
        c<-10+16;
                print(c);
c<-a-10;
        print(c);
c<-a+10;
        print(c);
c<--10+a;
        print(c);
        c<- -10+0;
print(c);
c<-a+b+15;
        print(c);
c<-(a+b)-15;
        print(c);

}
```

```
(*Test Case 2*)

int main()
{
        int a <- 100;
        int b <- 25;
        int c;

        c<-a << b;
                print(c);
        c<-b>>2;
                print(c);
        c<-b>>a;
                print(c);
        c<-a>>3;
                print(c);
        c<-a@b;
                print(c);
}

(*Test Case 3*)

int main()
{ int i;
        for (i <- 2 ;i < 5;i<-i+1)
                {
```

```
                print(i);
                }

                for(i <- 1 ;i < 10;i<-i * 2)
                {
                print(i);
                }

                for (i <- 10 ;i > 3 ; i<- i - 1)
                print(i);

        for (i<-0 ;i < 1; i <- 2 + i)
                print(i);
}

(*Test Case 4*)

int main()
{
        int a <- 100;
        int b <- 25;
        int c;

        c<-a>b;
                print(c);
```

```
}

(*Test Case 5*)

int main()
{
        int a <-100;
  int b <- 10;

        if(a<=b)
                print(a);
                else
                print(b);

        if(a>b)
                print(a);
        else
                {
                print(b);

                }

        if(a>=b+1)
                print(a);

        if(a>b || a<b)
```

```
                print(b);
        else
                print(a);

                if(a=b)
                print(a);
                else
                print(b);


}

(*Test Case 6*)

(# Declarations and Initializations #)

int main()
{
        int i,z;
        i<-8;
        int j <- 5;



        int a[10], b[3][3];
```

```
                for( z <- 1; z <= j;z <- z + 3)
                        {
                                a[j] <- z;

                                j<-j+1;
                        }
}
```

(*Test Case 7*)

```
int main()
{
        int a <- 100;
        int b <- 25;
        int c <- 25;

        a<b;
        b <= c;

        b<c ;
        a<-b<-c;
        a<-b!=b;
        a<=b!=b;
        (b!=b)>10;
```

```
        !a + b < c * 10;
        a/b*!20!=b;



        a>b;
        100>=99;
        b<a;
        (a-b)<b;
        (a+b)>=(b+a);
        a!=b;
        a!=100;
        25!=a;
                a<-b;

        2 ~mod 10;
        20 <> 30;
        a <> b;
}

(*Test Case 8*)

int main()
{

        int g, h<-2;
        int x<-10, y<-11, z<-5;
```

```
        int c[10],d[5][5];
        int a[]<-{1,2,3};
                int b[][3]<-{0,1,2;3,4,5;6,7,8};
        c[5]<-x;
        d[1][2]<-y;

        g<-c[5] * d[1][2] + h;
        print(g);
}

(*Test Case 9*)

int main()
{
  int i <- 0, a <- 5;
  while (i <= a)
  {
    print(i);
    i <- i + 1;
  }

  i <- 10;
  while ( i!=a)
        i <- i - 1;

  print(i);
```

```
 while(i>3)
{
        print(i);
        i<-i-1;
}

}

(*Test Case 10*)

int adder(int c, int a[], int b[][])
{
 int d;
 d<-c + a[1] * b[1][1];
 return d;
 }



int main()
{
        int a[]<-{1,2,3},c,e;
                int b[][3]<-{0,1,2;3,4,5;6,7,8};

        c <-2 * 5;
```

```
        e<-adder(c,a,b);
        print(e);
}
```

8. LESSONS LEARNED

8.1. HSIU-YU TSUANG

The first thing I learned from this course is how to make "Makefile". I have not heard of Makefile before taking this course. I also learned how to use the tool, OCamldep, to help us check the dependency rules through making Makefile.

I also learned how to write codes in OCaml. This language is actually the first functional language I have ever used (Lots of first-time thing from taking this course!). Programming in OCaml does gives me a different way of thinking about writing programs. And I personally found that OCaml is really indeed very handy for writing software systems like compiler. The reason being that doing a compiler requires lots of branching cases, which can be easily handled by the pattern matching features in OCaml language. In addition, many lists manipulation functions inbuilt in OCaml language makes dealing with function arguments and stack easy.

One more important thing is that I learned many software tools along with doing this project. I used MSYS with MinGW and VIM editor as my program development environment; I learned how to write shell scripts (although a very simple one) to make testing automation happen; also, I used SVN software to work on project files as a team.

8.2. MINITA SHAH

The initial period of programming in a functional language was tough, but once I crossed that barrier, developing the language became addictive. It was difficult to stop working on it until I finished what I intended to do. The entire project was an enjoyable experience and a great challenge. One of the important things it taught me was to keep my patience with a language I have never used before. Sometimes, debugging in Ocaml was frustrating but with time I understood what the errors meant and even avoided them. Since we were developing a compiler I had a chance to work with three languages, including Java, Ocaml and the language we were

developing and I realized how convenient it was to use a functional language to develop the compiler. Three lines of Ocaml code was better than numerous lines of Java code.

I also learnt the dynamics of working in a group and how important coordinating with the other team members was. A systematic approach towards managing the different files made life a lot easier and in the process I also learnt how to use Rapid SVN for version control.

8.3. SAKET GOEL

To begin with, making the project was a great learning experience in different aspects. The project made me aware of the challenges involved in making a compiler. It was the first time I used a functional language for programming. At first, I found it extremely difficult to get accustomed to it. But once I did, I realized its advantages and the great amount of reduction in the size of the code produced by it.

Group communication is very important. Once important aspect I learnt is that even if you are working in a group with modules divided among individuals, everyone should try to work in the same room. The way this helps is that if anyone is stuck up at any point for a long time, he can seek the help of the others in debugging and proceed soon rather than wasting too much time on an error. This helps immensely in increasing the speed at which the work is done. More over, if all the members of a team are equally dedicated, it motivates you to work that extra bit for the great amount of effort being put in by everyone. Having a team leader really helps. The project gave me an opportunity to use version control and other tools for the first time.

8.4. SARFRAZ NAWAZ

I was introduced to Functional programming due to this project. The Functional approach to programming initially did seem difficult to work with and debug, but in due course did turn out to be a more efficient means to code succinct programs in applications like compiler construction. The experience working in the project has instigated me to take up courses/projects related to language design and Embedded Systems in the coming semesters. Also, learned that a systematic approach and proper coordination in a team are key to delivery of a successful project within allotted time.

9. APPENDIX

9.1 SOURCE CODE

```
(* lexer.mll *)
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n']     { token lexbuf }
| "(#"                     { comment lexbuf }
| '+'                      { PLUS }
| '-'                      { MINUS }
| '*'                      { TIMES }
```

```
| '/'           { DIV }
| '%'           { MOD }
| '!'           { NOT }
| '<'           { LT }
| "<="          { LEQ }
| '='           { EQ }
| "!="          { NEQ }
| '>'           { GT }
| ">="          { GEQ }
| "&&"          {LAND}
| "||"          { LOR }
| '&'           {BAND}
| '|'           { BOR }
| '@'           { XOR }
| "~mod"        { MODINV }
| ">>"          {RSHIFT}
| "<<"          {LSHIFT}
| "<>"          { GCDCRYPS }
| '^'           { EXP}
| "int"         { INT }
| "string"      { STRING }
| "print"  { PRINT }
| "<-"          { ASSIGN }
```

```
| "if"              { IF }
| "else"            { ELSE }
| "for"             { FOR }
| "while"           { WHILE }
| "return"          { RETURN }
| ';'               { SEMI }
| '('               { LPAREN }
| ')'               { RPAREN }
| '{'               { LBRACE }
| '}'               { RBRACE }
| '['               { LSQUARE }
| ']'               { RSQUARE }
| ','               { SEQ }
| eof               { EOF }
| ['a'-'z' 'A'-'Z'] ['a'-'z' 'A'-'Z' '0'-'9' '_']* as id { ID(id) }
| ['0'-'9']+ as lit { LITERAL(int_of_string lit) }
| '"' ([^'"']* '"' as print_st)  { P_str(print_st) }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
"#)" { token   lexbuf }
|_   { comment lexbuf }
```

```
(* parser.mly *)
%{ open Ast %}


%token PLUS MINUS TIMES DIV MODINV ASSIGN RANDOM
%token LT LEQ EQ NEQ GT GEQ
%token SEQ SEMI LPAREN RPAREN LBRACE RBRACE LSQUARE
RSQUARE
%token MOD EXP IN TO GCDCRYPS
%token LAND LOR XOR BAND BOR
%token LSHIFT RSHIFT
%token NEG NOT
%token IF ELSE FOR WHILE INT VOID RETURN
%token <int> LITERAL
%token <string> ID
%token <string> P_str
%token EOF
%token INC PRINT STRING

%nonassoc NOELSE
%nonassoc ELSE

%left  ASSIGN
%left  SEQ
%left  LAND LOR XOR BAND BOR
%left  EQ NEQ
%left  LT LEQ GT GEQ GCDCRYPS
%right LSHIFT RSHIFT
%left  PLUS MINUS
%left  TIMES DIV MOD MODINV EXP
%left  NEG NOT

%start program
%type <Ast.program> program

%%

    program:
```

```
        block { $1 }


formal_arg_val:
        /*nothing*/   { [] }
        |formal_arg_list { List.rev $1 }

formal_arg_list:
formal_arg { [$1] }
| formal_arg_list SEQ formal_arg { $3 :: $1 }

formal_arg:
data_type ID { ($1,$2) }

block:
   stmt_list {Block (List.rev $1)}

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

stmt:
    expr SEMI { Expr($1) }
    | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
    | LBRACE block RBRACE { $2 }
    | RETURN expr SEMI { Return($2) }
    | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5,
                                                    block( []))}
    | IF LPAREN expr RPAREN  stmt ELSE stmt { If($3, $5, $7) }
    | FOR LPAREN expr SEMI expr SEMI expr RPAREN stmt
                                              {For($3, $5, $7, $9) }
    | data_type var_init_list SEMI { Init($1, List.rev($2)) }
    | var_assign SEMI { Var_assign($1) }
    | data_type ID LPAREN formal_arg_val RPAREN LBRACE
    stmt_list RBRACE{ Func($1, $2, $4, (List.rev $7)) }
    | PRINT LPAREN print_val_list RPAREN SEMI {
    Printl(List.rev($3)) }

print_val_list:
        print_val { [$1] }
        |print_val_list PLUS print_val { $3 :: $1 }
```

```
print_val:
        ID { Idp($1) }
        | P_str { Strlit($1) }

var_init_list:
        var_init { [$1] }
        |var_init_list SEQ var_init { $3 :: $1 }

var_init:
        ID { Ida($1)}
        |ID ASSIGN expr { Assigna($1, $3) }
        |ID LSQUARE LITERAL RSQUARE { Arr_init3($1, $3) }
        |ID LSQUARE LITERAL RSQUARE LSQUARE LITERAL
        RSQUARE { Mat_init3($1, $3, $6) }
        |ID LSQUARE RSQUARE ASSIGN digit_set{Arr_init4($1,$5)}
        |ID LSQUARE RSQUARE LSQUARE LITERAL RSQUARE
        ASSIGN digit_mat_set{ Mat_init4($1, $5, $8) }

var_assign:
        ID LSQUARE RSQUARE ASSIGN digit_set{Arr_assign($1,$5)}
        |ID LSQUARE RSQUARE LSQUARE LITERAL RSQUARE
        ASSIGN digit_mat_set{ Mat_assign($1, $5, $8) }

digit_mat_set:
        LBRACE digit_mat_list RBRACE { List.rev $2 }

digit_mat_list:
        digit_list { [$1] }
        | digit_mat_list SEMI digit_list { $3 :: $1 }

digit_set:
        LBRACE digit_list RBRACE { List.rev $2 }

digit_list:
        LITERAL { [$1] }
        | digit_list SEQ LITERAL { $3 :: $1}

data_type:
        INT    { "BigInteger" }
```

```
actual_arg_list:
        /* nothing*/   { [] }
        | actual_arg { List.rev $1 }

actual_arg:
        expr { [$1] }
        |actual_arg SEQ expr { $3 :: $1 }

expr:
         expr PLUS   expr { Binop($1, Add, $3) }
        | expr MINUS  expr { Binop($1, Sub, $3) }
        | expr TIMES  expr { Binop($1, Times, $3) }
        | expr MOD    expr { Binop($1, Mod, $3) }
        | expr MODINV expr { Binop($1, Modinv, $3) }
        | expr DIV    expr { Binop($1, Div, $3) }
        | expr EQ     expr { Binop($1, Eq, $3) }
        | expr NEQ    expr { Binop($1, Neq,$3) }
        | expr LT     expr { Binop($1, Less, $3) }
        | expr LEQ    expr { Binop($1, Leq, $3) }
        | expr GT     expr { Binop($1, Greater, $3) }
        | expr GEQ    expr { Binop($1, Geq, $3) }
        | expr LAND   expr { Binop($1, Land,$3) }
        | expr LOR    expr { Binop($1, Lor, $3) }
        | expr BAND   expr { Binop($1, Band,$3) }
        | expr BOR    expr { Binop($1, Bor, $3) }
        | expr XOR    expr { Binop($1, Xor, $3) }
        | expr LSHIFT expr { Binop($1, Lshift, $3) }
        | expr RSHIFT expr { Binop($1, Rshift, $3) }
        | expr EXP expr { Binop($1, Exp, $3) }
        | expr GCDCRYPS expr { Binop($1, Gcd, $3) }
        | LPAREN expr RPAREN { $2 }
        | MINUS expr { Unary(Neg, $2) }
        | NOT expr { Unary(Not, $2) }
        | ID ASSIGN expr { Assign($1, $3)}
        | LITERAL { Literal($1) }
        | ID { Id($1) }
        | ID LSQUARE expr RSQUARE { Arr($1, $3) }
        | ID LSQUARE expr RSQUARE LSQUARE expr RSQUARE
        {Mat($1, $3, $6) }
        | ID LSQUARE expr RSQUARE ASSIGN expr { Arr1($1, $3,
        $6)}
```

```
        | ID LSQUARE expr RSQUARE LSQUARE expr RSQUARE
        ASSIGN expr { Mat1($1, $3, $6, $9) }
        | ID LPAREN actual_arg_list RPAREN { Fun_call($1, $3) }


(*  ast.mli *)
        type binop = Add | Sub | Times | Div | Eq | Neq | Less | Leq |
        Greater | Geq |Mod | Xor | Lshift | Rshift | Land | Lor | Band
        | Bor |Exp | Gcd | Modinv

        type unary = Not | Neg


        type expr =
            Literal of int
          | Id of string
          | Binop of expr * binop * expr
          | Unary of unary * expr
          | Assign of string * expr
          | Arr of string * expr
          | Mat of string * expr * expr
          | Arr1 of string * expr * expr
          | Mat1 of string * expr * expr * expr
          | Fun_call of string * expr list

        type print_val =
                Idp of string
                | Strlit of string

        type var_init =
                Ida of string
                |Assigna of string * expr
                |Arr_init3 of string * int
                |Mat_init3 of string * int * int
                |Arr_init4 of string * int list
                |Mat_init4 of string * int * int list list

        type var_assign =
                |Arr_assign of string * int list
                |Mat_assign of string * int * int list list
```

```
        type stmt =
                  Block of stmt list
                | Expr of expr
                | While of expr * stmt
                | If of expr * stmt * stmt
                | Return of expr
                | For of expr * expr * expr * stmt
                | Init of string * var_init list
                | Var_assign of var_assign
                | Printl of print_val list
                | Func of string * string * (string * string) list * stmt list

        type program = stmt

(* printer.ml *)
open Ast

module NameMap = Hashtbl.Make(struct
type t = string
let equal x y =  x =  y
let hash = Hashtbl.hash
end)

exception ReturnException of int * int NameMap.t
let symboltable = ref [ ]
let functionTable = NameMap.create 128

let rec start_scope dummy =
let newSym = (NameMap.create 128) in
symboltable := newSym :: !symboltable;
" "
let rec end_scope dummy = match !symboltable with
    [] -> ""
    | hd :: tl -> symboltable := tl; ""

let varAdd name dType =
let head = List.hd !symboltable in
    if NameMap.mem head name then
       false
    else
       (NameMap.add head name dType; true)
```

```
let varFind name =
        let rec rec_varFind tbl =
           match tbl with
           [] -> 0
           | hd :: tl -> try NameMap.find hd name
             with Not_found -> rec_varFind tl
             in rec_varFind !symboltable


let varMem name =
let rec rec_varMem tbl =
   match tbl with
   [] -> false
   | hd :: tl ->
        if( NameMap.mem hd name) then true
           else rec_varMem tl
           in
           rec_varMem !symboltable

let rec typeof = function
Literal(l) -> "int"
| Id(s)      -> if varMem s then
                  if (varFind s = 1) then "int"
                  else if (varFind s = 2) then "Arr"
                  else if (varFind s = 3) then "Mat"
                  else (raise (Failure ("The Variable " ^ s ^ " is of invalid
                  type")))
                  else (raise (Failure ("Variable undeclared" ^ " " ^ s)))
| Binop(e1, op, e2) ->
let t1 = typeof e1 in
let t2 = typeof e2 in
if (t1 = t2) then t1
     else (raise (Failure ("Type mismatch in Binary expression ")))
| Arr(s, e)       -> if varMem s then
                     if (varFind s = 2) then "int"
                      else (raise (Failure ("Variable " ^ s ^ " used
                         incorrectly")))
                     else (raise (Failure (" Array undeclared" ^ " " ^ s)))
| Mat(s, e1, e2)   -> if varMem s then
                        if (varFind s = 3) then "int"
```

```
                        else (raise (Failure ("Variable " ^ s ^ " used
                        incorrectly")))
                        else (raise(Failure("Matrix undeclared" ^ " " ^ s)))
| Arr1(s, e1, e2) -> " "
| Mat1(s, e1, e2, e3) -> " "
| Unary (op, e)   ->  let t = typeof e in
                        (match t with
                        "int" -> "int"
| _ -> raise(Failure("Type mismatch in unary expression")))
| Assign(var, e)  ->  let t1 = "int" in
                        let t2 = typeof e in
                        if (t1 = t2) then t1
                        else (raise (Failure ("Type mismatch in assignment
                        expression ")))
| Fun_call("getPrime",len)-> "int"
| Fun_call("getRandom", bits)-> "int"
| Fun_call("getInt", bits)-> "int"
| Fun_call(f, actuals) -> let getActualsTypeList actuals = List.map
typeof actuals in
                            if NameMap.mem functionTable f then
                            let check_arglist actuals =
                            let (numArgs, typeList) = NameMap.find
                            functionTable f in
                            let actualsTypeList = getActualsTypeList
                            actuals in
let boolV = List.fold_left2 (fun boolValue type1 type2 ->
if (type1= type2) || ((type1= "int") && (type2="BigInteger")) then
boolValue  else false ) true actualsTypeList typeList in
if((numArgs != (List.length actuals)) || (not boolV))
then raise (Failure ("Formal argument list of " ^ f ^ " does not
match actual argument list"))
else ignore("")in check_arglist actuals; "int"
   else
      raise (Failure ("Function not defined" ^ f))


let rec string_of_expr = function
Literal(l)          -> "(new BigInteger(" ^ "\"" ^ string_of_int l ^ "\"))"
| Id(s)             -> if varMem s then s
                        else raise (Failure ("undeclared identifier fdsf" ^ s));
```

```
| Arr(s, e)         -> if varMem s then s ^ "[(" ^ string_of_expr e ^
").intValue()]"
                        else raise (Failure ("Undeclared variable " ^ s));
| Mat(s, e1, e2)  -> if varMem s then s ^ "[(" ^ string_of_expr e1 ^
").intValue()]" ^ "[(" ^ string_of_expr e2 ^ ").intValue()]"
                        else raise (Failure ("Undeclared variable " ^ s));
| Arr1(s, e1, e2) -> if varMem s  then s ^ "[(" ^ string_of_expr e1 ^
").intValue()]" ^  " = " ^string_of_expr e2
                       else raise (Failure ("Undeclared variable " ^ s));
| Mat1(s, e1, e2, e3) -> if varMem s then s ^ "[(" ^ string_of_expr e1 ^
").intValue()]" ^ "[(" ^ string_of_expr e2 ^ ").intValue()]" ^
" = " ^ string_of_expr e3
                          else raise (Failure ("Undeclared variable " ^ s));
| Binop(e1, o, e2) -> (match o with
                        Add     -> string_of_expr e1 ^ "." ^ "add" ^ "(" ^
                        string_of_expr e2 ^ ")"
                        | Sub     -> string_of_expr e1 ^ "." ^ "subtract" ^ "("
                        ^ string_of_expr e2 ^ ")"
                        | Times   -> string_of_expr e1 ^ "." ^ "multiply" ^ "("
                        ^ string_of_expr e2 ^ ")"
                        | Div     -> string_of_expr e1 ^ "." ^ "divide" ^ "(" ^
                        string_of_expr e2 ^ ")"
                        | Mod     -> string_of_expr e1 ^ "." ^ "mod" ^ "(" ^
                        string_of_expr e2 ^ ")"
                        | Modinv  -> string_of_expr e1 ^ "." ^ "modInverse"
                        ^ "(" ^ string_of_expr e2 ^ ")"
                        | Exp     -> string_of_expr e1 ^ "." ^ "pow" ^ "(" ^
                        string_of_expr e2 ^ ".intValue()" ^ ")"
                        | Eq      -> "(" ^ string_of_expr e1 ^ "." ^
                        compareTo" ^ "(" ^ string_of_expr e2 ^ ")" ^ "== 0 ?
                        true : false)"
                        | Neq     -> "(" ^ string_of_expr e1 ^ "." ^
                        "compareTo" ^ "(" ^ string_of_expr e2 ^ ")" ^ "!= 0 ?
                        true : false)"
                        | Less    -> "(" ^ string_of_expr e1 ^"." ^compareTo"
                        ^ "(" ^ string_of_expr e2 ^ ")" ^ "< 0 ? true : false)"
                        | Leq     -> "(" ^ string_of_expr e1 ^ "." ^compareTo"
                        ^ "(" ^ string_of_expr e2 ^ ")"  ^ "<= 0 ? true : false)"
                        | Greater -> "(" ^ string_of_expr e1 ^ "." compareTo"
                        ^ "(" ^ string_of_expr e2 ^ ")" ^ "> 0 ? true : false)"
```

```
                    | Geq     -> "(" ^ string_of_expr e1 ^ "."^"compareTo"
                    ^ "(" ^ string_of_expr e2 ^ ")" ^ ">= 0 ? true : false)"
                    | Land    -> string_of_expr e1 ^ " " ^ "&&" ^ " " ^
                    string_of_expr e2
                    | Lor     -> string_of_expr e1 ^ " " ^ "||"  ^ " " ^
                    string_of_expr e2
                     | Band    -> string_of_expr e1 ^ "." ^ "and" ^ "(" ^
                    string_of_expr e2 ^ ")"
                    | Bor     -> string_of_expr e1 ^ "." ^ "or"  ^ "(" ^
                    string_of_expr e2 ^ ")"
                    | Xor     -> string_of_expr e1 ^ "." ^ "xor" ^ "(" ^
                    string_of_expr e2 ^ ")"
                    | Lshift  -> string_of_expr e1 ^ "." ^ "shiftLeft"  ^ "("
                    ^ string_of_expr e2 ^ ".intValue())"
                  | Rshift  -> string_of_expr e1 ^ "." ^ "shiftRight" ^ "(" ^
                  string_of_expr e2 ^ ".intValue())"
                  | Gcd     -> string_of_expr e1 ^ "." ^ "gcd" ^ "(" ^
                  string_of_expr e2 ^ ")")
                  | Assign(var, e)  ->  var ^ " = "  ^ string_of_expr e
                  | Unary (op, e)   ->  string_of_expr e ^ "." ^(match op
                   with
                  Neg     -> "negate"
                  |  Not     -> "not") ^ "(" ^ " " ^ ")"
                  |  Fun_call("getPrime",len)-> getPrime(" ^
               String.concat ", " (List.map string_of_expr len) ^ ")"
                  |  Fun_call("getRandom", bits)-> let getInt num =
               string_of_expr num ^ ".intValue()" in
                 "new BigInteger" ^ "(" ^ String.concat ", " (List.map
                 getInt bits) ^ "," ^ "new Random())"
                  |  Fun_call(f, el) -> f ^ "(" ^ String.concat ", " (List.map
                 string_of_expr el) ^ ")"


let string_of_formal_arg = function (t, id) -> t ^ " " ^ id
let string_of_data_type  = function s -> s


let string_of_var_init = function
  Ida(s)->
      let v = 1 in
```

```
        ignore(if NameMap.mem (List.hd !symboltable) s then raise (Failure
        ("ID already initialized " ^ s))
                else varAdd s v);
        " " ^ s ^ " = new BigInteger(\"0\")"
 | Assigna(v, e) -> (let m = 1 in
 ignore(if NameMap.mem (List.hd !symboltable) v then raise (Failure ("ID
already initialized " ^ v))
        else varAdd v m);
        " " ^ v ^ " = " ^ string_of_expr e)

 | Arr_init3 (s, l)   -> let v = 2 in
  ignore(if NameMap.mem (List.hd !symboltable) s then raise (Failure
("Variable already initialized " ^ s))
        else varAdd s v);
        " " ^ s ^ "[" ^ "] " ^ " = " ^ "new BigInteger " ^ "[" ^ string_of_int l ^ "]"
 | Mat_init3 (s, l1, l2)   -> let v = 3 in
ignore(if NameMap.mem (List.hd !symboltable) s then raise (Failure ("Variable
already initialized " ^ s))
        else varAdd s v);
        " " ^ s ^ "[" ^ "]" ^ "[" ^ "] " ^ " = " ^ "new BigInteger " ^ "["
        ^string_of_int l1 ^ "]" ^ "[" ^ string_of_int l2 ^ "]"
 | Arr_init4 (s, d) -> let v = 2 in
  let arrStr i = "new BigInteger(" ^ "\"" ^ string_of_int i ^ "\")" in
  ignore(if NameMap.mem (List.hd !symboltable) s then raise (Failure
("Variable declared with diff size " ^ s))
        else varAdd s v);
        " " ^  s ^ "[" ^ "]" ^ " = " ^ "{ " ^ String.concat "," (List.map arrStr d) ^"}"
 | Mat_init4 (s, l, d) -> let v = 3 in
 let arrStr i = "new BigInteger(" ^ "\"" ^ string_of_int i ^ "\")" in
 ignore(if NameMap.mem (List.hd !symboltable) s
 then raise (Failure ("Variable declared with diff size " ^ s))
        else varAdd s v);
        " ^ s ^ "[" ^ "]" ^"[" ^ string_of_int l ^ "] " ^ " = " ^ "{ " ^ String.concat ","
        (List.map( fun f -> "{" ^ String.concat "," (List.map arrStr f) ^ "}") d) ^
        "}"

let string_of_var_assign = function
Arr_assign (s, d) -> if varMem s then " " ^ s ^ "[" ^ "] " ^ " = " ^ "{ " ^
String.concat "," (List.map string_of_int d) ^ " }"
                        else raise (Failure ("Variable undeclared" ^ s))
```

```
 | Mat_assign (s, l, d) -> if varMem s then " " ^ s ^ "[" ^ "]" ^"[" ^ string_of_int l
^ "] " ^ " = " ^ "{ " ^ String.concat "," (List.map( fun f -> "{" ^ String.concat ","
((List.map string_of_int f)) ^ "}") d) ^ " }"
                              else raise (Failure ("Matrix undeclared " ^ s))

let string_of_printval = function
Idp(s)          -> s
| Strlit(stl)   -> "\"" ^ stl
                    let rec string_of_stmt = function
                    Block(stmts)    ->ignore( start_scope "startBlock");
                   let str = "\n" ^ String.concat "" (List.map string_of_stmt (stmts
                   ))  in
                   end_scope "endBlock"^"\n" ^ str;

| Expr (expr)        -> ignore(typeof expr); string_of_expr expr ^ ";\n"
| Return(expr)       -> if (typeof expr <> "int") then raise(Failure("Type
 mismatch in Return statement")) else  "return " ^ string_of_expr expr ^ ";\n";
| If(e, s, Block([])) -> if (typeof e <> "int") then raise(Failure("Type
 mismatch in If statement")) else "if (" ^ string_of_expr e ^ ")\n{" ^
string_of_stmt s ^ "\n}"
| If(e, s1, s2)      -> if (typeof e <> "int") then raise(Failure("Type
 mismatch in If statement")) else "if (" ^ string_of_expr e ^ ")\n{" ^
 string_of_stmt s1 ^ "\n}"^"else\n{" ^ string_of_stmt s2 ^ "\n}"
| For(e1, e2, e3, s)  -> if (typeof e1 <> "int" || typeof e2 <> "int" ||
 typeof e3 <> "int") then raise(Failure("Type mismatch in For statement"))
else "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^ string_of_expr
e3 ^ ")\n{" ^ string_of_stmt s ^ "\n}"
| While(e, s) -> if (typeof e <> "int") then raise(Failure("Type mismatch in
While")) else " while (" ^ string_of_expr e ^ ")\n{" ^ string_of_stmt s ^ "\n}"
| Func (s1, s2, l1, l2) -> ignore(start_scope "startBlock");
 let v = 4 in
           if s1 <> "BigInteger" then raise (Failure ("Invalid return type: " ^ s2))
                 (*First we check the return type of function definition*)
            else begin      (*If it is valid, we do the following tasks: *)
           if NameMap.mem functionTable s2 (*Test if function name is already
           then raise (Failure ("Function already defined: " ^ s2)) defined*)
           else begin
           ignore(varAdd s2 v); (*It is not yet defined; so we add fxn name into
           current symbol table*)
           let typeList = List.map (fun (mytype, name) -> mytype) l1 in
           NameMap.add functionTable s2 ((List.length l1), typeList )  ;
```

```
                List.iter ( fun (t, id) -> if t = "BigInteger" then ignore (varAdd id 1)
                 else if   t = "Arr" then ignore (varAdd id 2)
                 else if   t = "Mat" then ignore (varAdd id 3)
                 else raise (Failure (id ^ ": Invalid argument type in "
                 ^ s2)); ) l1;
                if s2="main"
                then let str = "public static void main(String[] args)\n" ^
                "{\n " ^ String.concat " " ((List.map string_of_stmt l2)) ^
                "\n}\n" in end_scope "endFunc" ^ str;
                else "public static" ^ " " ^ s1 ^ " " ^ s2 ^
                let str = "(" ^ String.concat "," ((List.map string_of_formal_arg l1)) ^
               ")\n" ^
               "{\n " ^ String.concat " " ((List.map string_of_stmt l2)) ^ "\n}\n"
                in end_scope "endFunc" ^ str;
                end
               end
  | Init(d,v)    ->  d ^ String.concat "," ((List.map string_of_var_init v ))^ ";\n"
  | Var_assign(v) ->  (string_of_var_assign v)^";\n"
  | Printl(str) ->
   "System.out.println(" ^ String.concat "+" (List.map string_of_printval str) ^ ")" ^
";\n"

 let string_of_vdecl id = "BigInteger" ^ " " ^ id ^ ";\n"

let string_of_program stmt = ( "package test;\nimport
java.math.BigInteger;\nimport java.util.Random;\npublic class Cryps\n{\nprivate
static final Random rnd = new Random();
                private static boolean miller_rabin_pass(BigInteger a, BigInteger n)
                {
                   BigInteger n_minus_one = n.subtract(BigInteger.ONE);
                   BigInteger d = n_minus_one;
                   int s = d.getLowestSetBit();
                   d = d.shiftRight(s);
                   BigInteger a_to_power = a.modPow(d, n);
                   if (a_to_power.equals(BigInteger.ONE)) return true;
                   for (int i = 0; i < s-1; i++)
                         {
                      if (a_to_power.equals(n_minus_one)) return true;
                      a_to_power = a_to_power.multiply(a_to_power).mod(n);
                   }
                   if (a_to_power.equals(n_minus_one)) return true;
```

```
                return false;
        }


        public static boolean miller_rabin(BigInteger n)
        {
           for (int repeat = 0; repeat < 20; repeat++)
                {
              BigInteger a;
              do
                        {
                 a = new BigInteger(n.bitLength(), rnd);
              } while (a.equals(BigInteger.ZERO));
              if (!miller_rabin_pass(a, n))
                        {
                 return false;
              }
           }
           return true;
        }

        public static BigInteger getPrime(BigInteger len)
        {

              int nbits = len.intValue();
           BigInteger p;
           do
                {
                p = new BigInteger(nbits, rnd);
                }
                while (!miller_rabin(p));
                return(p);
         }\n" ^ string_of_stmt (stmt)) ^ "}"  ;


(* run.ml *)

let compile = true

let _ =
        let lexbuf  = Lexing.from_channel stdin in
```

```
        let program = Parser.program Lexer.token lexbuf in
        if  compile then
           let listing = Printer.string_of_program program in
           print_string listing
```

```
(*Makefile*)

# This is makefile version 1.0

# define MACRO first
OBJECTS = lexer.cmo parser.cmo printer.cmo run.cmo # these are the objects files
that will be created
OCAMLC  = ocamlc                    # ocamlc just like gcc in c

#Top level build target
run: Makefile $(OBJECTS)
        $(OCAMLC) -o run $(OBJECTS)

# building target: cryps
#cryps:  $(OBJECTS)
#       $(OCAMLC) -o cryps $(OBJECTS)

# dependency rules for all object files
%.cmo: %.ml
        $(OCAMLC) -c $<

# dependency rules for all .ml files
lexer.ml: lexer.mll
        ocamllex lexer.mll

parser.ml parser.mli: parser.mly
```

```
	ocamlyacc parser.mly

# dependency rules for all .cmi files ; .cmi files is generated from interface files(.mli)
by ocamlc
%.cmi: %.mli
	$(OCAMLC) -c $<

.PHONY : clean
clean:
	rm -f lexer.ml parser.ml parser.mli *.cmo *.cmi

# Dependency rules generated by ocamldep *.ml *.mli. Thanks for ocamldep for
checking dependency for us
lexer.cmo: parser.cmi
lexer.cmx: parser.cmx
parser.cmo: ast.cmi parser.cmi
parser.cmx: ast.cmi parser.cmi
printer.cmo: ast.cmi
printer.cmx: ast.cmi
parser.cmi: ast.cmi
run.cmo: printer.cmo parser.cmi lexer.cmo
run.cmx: printer.cmx parser.cmx lexer.cmx
```