fCryptica:利用マニュアル

2012/03/03

目 次

1　公開鍵暗号の実装を目的としたフレームワーク

1.1　仕様決定のための原則

1.1.1　ユーザ像の設定

設計にあたり，まず考慮しなくてはならないのが公開鍵暗号の実装を目的としたフレームワーク (以下，単にフレームワークと呼ぶ) をどんな人に使ってもらうかを想定することである．すなわち，ター ゲットとなるユーザを決めなくてはならない．

フレームワークを利用したいと考える人は，用意された暗号アルゴリズムの動作テストを行うことを目的としている．暗号アルゴリズムを考える人は数学家であり，暗号アプリケーションを作る人はプログラマである．動作テストが行われたアルゴリズムは数学家からプログラマへとコードとして渡され，最終的にはそのコードを元に暗号アプリケーションの開発へと繋がるだろう．

したがって，ユーザは暗号研究をしている数学家自身か，あるいは暗号の勉強をしようとしている数学系の学生であると考えられる．すなわち，全くの素人ではなく，ある程度の知識を有しているような人物が対象となる．ここで言うある程度の知識とは，既存の暗号アルゴリズムを理解し，別の数学家に解説ができるレベルの知識を指す．

対象となるユーザ
: 暗号研究をしている数学家自身か，あるいは暗号の勉強をしようとしている数学系の学生であり，既存の暗号アルゴリズムを他の数学家に解説できるレベルの知識を有する者．

1.1.2　目的と原則

フレームワークを提供する目的は，暗号研究の促進によって，将来危惧されている「量子コンピュータの登場による現代暗号の解読」を防ぐことである．これを大きな目的と呼ぶことにする．

一方で，ツールを利用したいと考えているユーザは，なるべく自分の手間をかけずに暗号アルゴリズムを記述し，動作確認をしたいと思っている．自分が考案した暗号アルゴリズムが簡単に実装できれば良いわけで，その先にあるプログラマへの伝達コストは必ずしも考慮されていない．したがって，ユーザにとっての目的は簡単に暗号アルゴリズムが記述できることである．これを小さな目的と呼ぶことにする。

まず，大きな目的を達成するためには，フレームワークが広く普及し，暗号研究をしている数学家および暗号の勉強をしようとしている数学系の学生の多くがフレームワークを利用することが必要となる．そのためには，次のような条件が必要であると考えられる．

1. 実行環境になるべく依存しない
   ユーザの実行環境に依存していると，フレームワーク普及の妨げとなる．フレームワークは大多数のユーザが使用していると思われる実行環境 (Windows，Linux，Mac など) で

実行できるようになっているか，もしくは実行環境に依存しにくいクロスプラットフォーム[1]なもの (Java や Web など) であることが望ましい．

2. 簡単に導入できる
   普及の観点から，簡単に配布可能であることは重要である．フレームワークは Web 上にアップロードされ，誰でも入手出来るようになっていることが望ましい．単にライブラリだけの配布は，利用するプログラム言語の開発環境をあらかじめ構築していることが前提となる．普及の観点から見ると，そのようなフレームワークは望ましくない．そのためフレームワークのインストール以外に必要となるものは，なるべく少なくする必要がある．

また，小さな目的を達成するためには，暗号アルゴリズムを実装し簡単に動作確認を行うまでに，なるべく手間のかからないフレームワークが必要となる．そのためには，次のような条件が必要であると考えられる．

1. 暗号アルゴリズムを記述するための情報を入力する以外の操作は少なくする
   暗号アルゴリズムの記述にあたり，鍵や平文の入力は避けることが出来ない．したがって，それ以外の部分においてユーザが行う操作をなるべく少なくする必要がある．

2. 暗号アルゴリズムの記述だけでなく，実際に実行したときの出力表示まで用意する
   実際に暗号アルゴリズムの動作確認を行う段階では，出力結果が見られる状況になっていなければならない．しかし従来のライブラリでは，出力形式もユーザ自身が記述する必要があったため手間がかかる．不必要な入力を減らし，動作確認が行えるフレームワークが望ましい．

3. ユーザが記述したコードを一つのプログラムとして出力する
   最終的に，動作確認後の暗号アルゴリズムは数学家からプログラマへと受け渡される．このとき，プログラムとして暗号アルゴリズムをプログラマに伝えることで，理解にかかる時間を大幅に減らすことができる．また，プログラムの一部を流用することもできる．

4. 別のフレームワークは必要無いようにする
   そのフレームワークを用いれば，どんな暗号アルゴリズムも記述でき，動作確認が行えるという状況が理想である．しかし，フレームワークを提案するきっかけとなった暗号ライブラリでさえ，発表されているすべての暗号アルゴリズムを記述できるわけではない．かといって，暗号アルゴリズムに応じて必要な数値計算を新たに実装することは煩わしい．必要な数値計算を新たに実装する必要がないように，なるべく多くの数値計算に対応することで，ユーザの様々な要求に答えることができる．

これらの原則が守られていれば，良いフレームワークであると考えられる．次節でこれらの原則をもとに，仕様を決定していく．

[1]異なるプラットフォーム (仕様が全く異なる機械または OS) 上で，同じ仕様のものを動かすことが出来るプログラム (ソフトウェア)

1.2 原則をもとにした仕様の設計

原則をもとに，実際にどういう設計をしたら良いのかを Table1.1 に示す．大きな目的によって定められた原則を B1,B2，小さな目的によって定められた原則を S1,S2,S3,S4 として表記し，どの原則に基づいて定められた仕様なのかを明らかにしている．

Table 1.1: 原則をもとにした仕様の設計

No.	仕様	基づく原則
1	Java などのクロスプラットフォームな言語で開発されたもの	B1
2	ソフトウェアは Web 上で利用可能か，ダウンロード可能な環境におく	B2
3	それだけで実行可能な形式をとる	B2
4	入力欄へ情報を記述することで動作確認ができる形式をとる	S1
5	記述の際の柔軟性に優れている言語の利用	S1
6	動作確認を行った際に結果が出力される形式をとる	S2
7	一つのプログラムファイルとして出力できる形式をとる	S3
8	なるべく多くの数値計算機能を搭載する	S4
9	なるべく多くのプログラミング言語に対応している	S4

1,2,3,4 は原則通りであるので，説明は省略する．それ以降の 5,6,7,8,9 について，なぜこのような仕様が必要になるのかそれぞれ説明する．

1.2.1 柔軟性の優れた言語の利用

暗号アルゴリズムを記述するユーザは数学者であると想定している．そのため，暗号アルゴリズムの記述以外を少なくすることは当然であるが，暗号アルゴリズムの記述も数学的な記述に近いものであることが求められる．

利用するプログラミング言語の仕様で表現の方法が制限されている場合，数学的な記述に近いものにできないことがある．例として，Java による記述ではベクトルによる加算が次のように表される．

Java によるベクトル演算 (加算の例)

```
// ベクトル型の A と B と C が定義されているとする
// ベクトルの加算 C=A+B
C = A.add(B)
```

数学的記述に近づけるのであれば，上記の場合 C=A+B と記述するのが望ましい．

簡単な例で示したが，プログラミング言語により記述の柔軟性が制限される．想定しているユーザが数学者である以上，数学的な記述に近い記述を可能とする柔軟性に優れた言語が要求される．そのため，新たに柔軟性に優れた言語を開発するか，既存の言語で柔軟性に優れたものを採用する必要がある．

1.2.2 動作確認のための結果出力機能

数学家が暗号アルゴリズムの動作確認までを行うことを想定している．また，暗号で扱われる結果の多くは長い数字列である．そのため，途中の値を簡単に出力できるようにする必要がある．

1.2.3 プログラム出力機能

フレームワークの目的の一つは数学家が暗号アルゴリズムを簡単に動作確認できることである．しかし，最終的にはプログラマに伝えるべきものであるため，プログラムとして渡されることが望ましい．そのため，原則 $B1$ に示した「暗号アルゴリズムを記述するための情報を入力する以外の操作は少なくする」を損なうことなく，プログラムを出力する機能が必要である．

1.2.4 様々な数値計算への対応

現在研究されている公開鍵暗号に利用される数学的問題を 3 章 3 節で示した．その問題から考えられる必要な数値計算機能を Table1.2 に示す．なお，数学的問題である素因数分解問題，離散対数問題，ナップザック問題，多次多変数多項式の求解問題，線形符号の復号問題，および巡回セールスマン問題を，簡単のためにそれぞれ P1,P2,P3,P4,P5,P6 と表記する．

Table 1.2: 暗号アルゴリズムにおける数学的問題と数値計算

数値計算	P1	P2	P3	P4	P5	P6
多倍長整数演算	○	○	○	○	○	○
有限体演算	○	○	○	○	○	○
複素数演算	-	-	-	-	-	○
有理数演算	-	-	-	○	○	-
一変数多項式演算	-	-	-	○	-	-
多変数多項式演算	-	-	-	○	-	-
ベクトル演算	-	-	○	○	○	-
行列演算	-	-	-	○	○	-

また，既存の暗号ライブラリにおける数値計算への対応は，Table1.3 に示す通りになっている．なお，4 章で紹介したオープンソースな暗号ライブラリである OpenSSL，Crypto++，NTL，Java 標準ライブラリ (J2SE)，および BouncyCastle(BC) の対応状況を示す．

フレームワークを開発するにあたり，少なくともこれ以上の数値計算機能に対応していることが求められる．また，現在研究されている暗号アルゴリズムにおける数学的問題に対応した数値計算機能を可能な限り提供する必要がある．

Table 1.3: 既存の暗号ライブラリが持つ数値計算機能

数値計算	OpenSSL	Crypto++	NTL	J2SE	BC
多倍長整数演算	○	○	○	○	○
有限体演算	○	○	○	○	○
複素数演算	-	-	○	-	-
有理数演算	-	-	○	-	-
一変数多項式演算	-	-	○	-	-
多変数多項式演算	-	-	-	-	-
ベクトル演算	-	-	○	-	-
行列演算	-	-	○	-	-

1.2.4.1 多倍長整数

一般に，計算機上では基本演算の対象であるデータの長さ (語長) が決まっている (32bit，64bit 等)．1 語が 32bit ならば 0 以上 232 未満の数しか表現できない．しかし語が複数集まることによりそれ以上の数を表現できるようになる．

ここで 1 語が $n$bit とすると，一般の正整数 A は，

$$A = A_m B^{m-1} + A_{m-1} B^{m-2} + ... + A_2 B + A_1$$

$$B = 2^n, 0 \leq A_i < B, A_m \neq 0$$

と書くことができ，m 個の語 $A_m, A_{m-1}, ..., A_1$ として計算機内で扱うことができるようになる．このように表現された整数を多倍長 (精度) 整数という．それに対して 1 語で表現できる整数を単精度整数または固定長 (精度) 整数ということがある．

暗号で扱う整数は巨大な整数であることが多く，多倍長整数を扱えることが必要不可欠である．

1.2.4.2 複数の要素を持つ数

整数をただ一つの要素からなる数と定義したとき，複素数は実部と虚部からなる数，有理数は分母と分子からなる数として表現される．また，多項式は単項式の集合であり，その単項式は次数と係数および変数で構成される式である．

しかし，数学家は数や式を記述するとき，複素数，有理数，多項式をそれぞれ一つの要素として扱う．そのため，複数の要素を持つ数それぞれの型を定義し，一つの要素として扱うことができる必要がある．

1.2.4.3 ベクトルと行列

数学におけるベクトルと行列は数または式の集合として扱われる．すなわち，整数，複素数，有理数，多項式を要素として持つ．

しかし，数学家はベクトルまたは行列自体を一つの要素として扱う．そのため，内部的に数や式を要素として持つ型を定義し，一つの要素として扱うことができる必要がある．また，行列をベクトルを要素として持つベクトルとして表現することもあり，それを許す型として定義する必要がある．

1.2.4.4　有限体演算

暗号では，有限体を利用した数値計算が多く使われる．そのため，暗号で扱われる整数，複素数，有理数，多項式，ベクトル，行列すべてにおいて有限体上での演算が可能となる関数を用意する必要がある．

1.2.5　様々なプログラミング言語への対応

数学家がフレームワークを用いて実装した暗号アルゴリズムは，最終的にプログラマに伝えられる．そのため，プログラマが利用するプログラミング言語に対応した出力が可能であることが必要である．

1.3　fCryptica の開発

本研究では，5 章 2 節で示した仕様に基づき，fCryptica というフレームワークの開発を行った．起動時の画面は Figure1.1 のようになる．

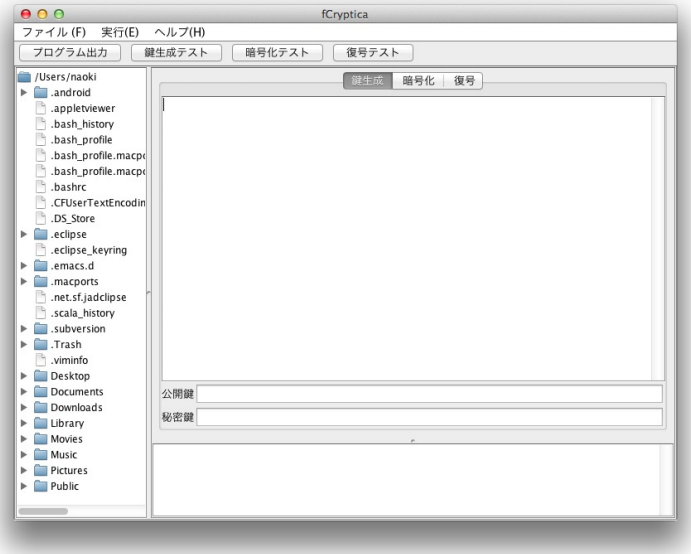

Figure 1.1: fCryptica 起動画面

画面最上部にフレームワークのメニューバーを用意しており，以下の操作を行うことができる．

- ファイル

  新規 新規に空のファイルを作成し，開く
  開く 既存のファイルを開く
  上書き保存 ファイルを上書き保存する
  別名で保存 ファイルを別名で保存する
  終了 fCryptica を終了する

- 実行

  プログラム出力 入力された情報をプログラムファイルとして出力する
  鍵生成テスト 入力された鍵生成アルゴリズムの動作確認を行う
  暗号化テスト 入力された暗号化アルゴリズムの動作確認を行う
  復号テスト 入力された復号アルゴリズムの動作確認を行う

- ヘルプ

  fCryptica について fCryptica のバージョン情報を表示する

また，そのすぐ下には暗号アルゴリズムの動作確認を行うための機能と，プログラムファイルとして出力するためのプログラム出力機能がショートカットボタンとして用意されている．

画面左側には，ファイルやフォルダを管理するエクスプローラが用意されており，ファイルをドラッグ＆ドロップで簡単に展開できるようにしている．画面右側は入出力画面としている．

このフレームワークを利用し，次の 13 ステップで公開鍵暗号の開発を行うことができる．

1. 鍵生成アルゴリズムの入力
2. 公開鍵，秘密鍵の指定
3. 鍵生成アルゴリズムの動作確認
4. 暗号化アルゴリズムに利用する鍵の指定
5. 平文の入力
6. 暗号化アルゴリズムの入力
7. 暗号文の指定
8. 暗号化アルゴリズムの動作確認
9. 復号アルゴリズムに利用する鍵の指定
10. 復号アルゴリズムの入力
11. 復号文の指定
12. 復号アルゴリズムの動作確認
13. プログラムファイルとして出力

この流れに沿っていくだけで，公開鍵暗号のエンジンとなる部分の実装が完成に至る．完成まで至れば，出来あがった暗号アルゴリズムがプログラムファイルとして出力されるので，あとはプログラマにそのファイルを渡すだけでよい．さて，仕様に対してどのような設計で実現したのかを示していく．

1.3.1 実行環境

fCryptica は Scala[?] で開発されているので，Java Runtime Environment(JRE)6.0 以上がインストールされている環境ならば動作する．

Scala は記述の際に柔軟性の優れている言語 (Robert らの文献 [?] により柔軟性が評価されている) である．この柔軟性を用いてライブラリを設計することで，暗号に必要な数値計算用の関数はより数式に近い記述で利用可能となる．

また，Scala は既存の Java プログラムとも容易に連携させることができる．そのため，数学家が Scala で実装した暗号アルゴリズムを Java ユーザはそのまま利用することが可能である．

1.3.2 導入

https://github.com/BuchoHack/fCryptica からダウンロードすることができる．

実行ファイル (jar) 形式なので，特にインストールする必要はない．ダウンロードしたファイルを任意のディレクトリに置き，fCryptica.jar ファイルを実行すると起動できる．

1.3.3 入力形式

情報の入力は鍵生成に関するステップ 1〜2，暗号化に関するステップ 4〜7，および復号に関するステップ 9〜11 で行われる．まず，鍵生成に関する情報の入力は Figure1.2 のようになっている．

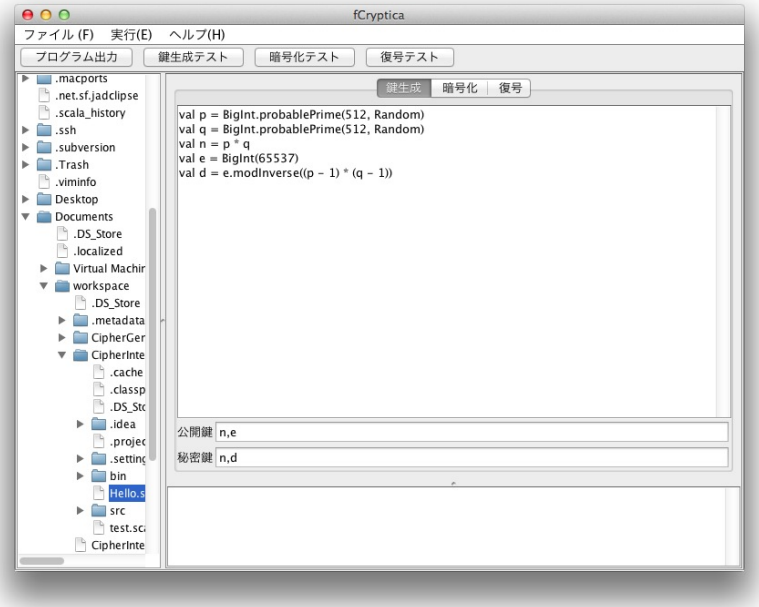

Figure 1.2: fCryptica 鍵生成アルゴリズム入力画面

この画面は，RSA 暗号の鍵生成アルゴリズムを入力した画面である．鍵生成アルゴリズムの入力，および公開鍵と秘密鍵の指定を行なう形式をとっている．

- 公開鍵と秘密鍵の指定
  公開鍵と秘密鍵それぞれに対して，その鍵が持つ情報が何であるかを入力し，指定した公開鍵と秘密鍵を，暗号化アルゴリズムおよび復号アルゴリズムで利用可能とする．

次に，暗号化に関する情報の入力は Figure1.3 のようになっている．この画面への遷移は入力画面の上タブをクリックすることで変更できる．

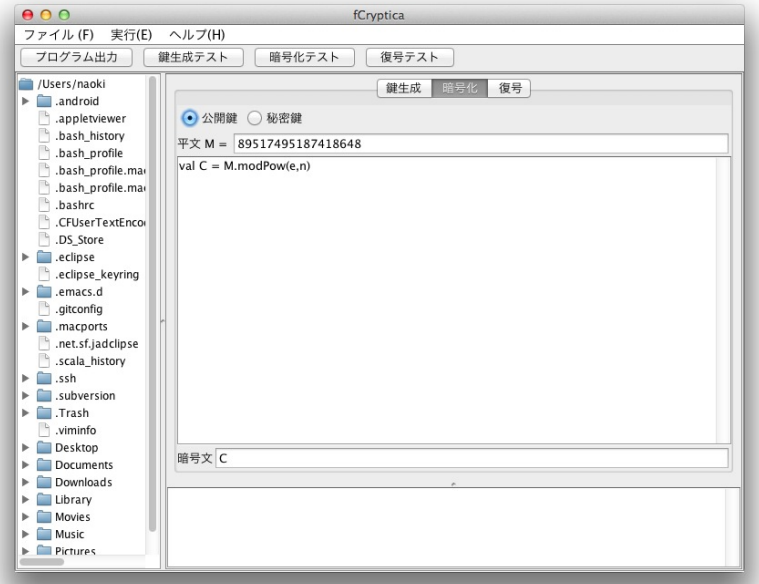

Figure 1.3: fCryptica 暗号化アルゴリズム入力画面

この画面は，RSA 暗号の暗号化アルゴリズムを入力した画面である．利用する鍵の選択，平文の入力，暗号化アルゴリズムの入力，および暗号文の指定を行なう形式をとっている．

- 鍵の選択
  鍵を選択することで，その鍵が持っている情報を利用できる．鍵が持っている情報とは，鍵生成アルゴリズムで入力した公開鍵または秘密鍵が持つ情報を指す．
- 平文の入力
  動作確認の時に必要となるパラメタである．
- 暗号文の指定
  暗号文が持つ情報が何であるかを入力し，指定した暗号文を復号アルゴリズムで利用可能とする．

最後に，復号に関する情報の入力は Figure1.4 のようになっている．画面の遷移は，前ステップと同様に入力画面の上タブをクリックすることで変更できる．

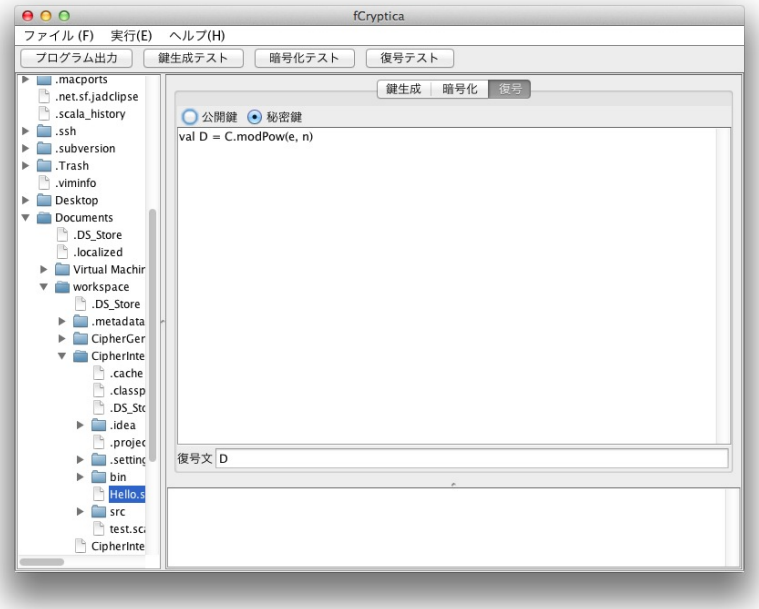

Figure 1.4: fCryptica 復号アルゴリズム入力画面

この画面は，RSA 暗号の復号アルゴリズムを入力した画面である．利用する鍵の選択，暗号化アルゴリズムの入力，および暗号文の指定を行なう形式をとっている．

- 鍵の選択
  鍵を選択することで，その鍵が持っている情報を利用できる．鍵が持っている情報とは，鍵生成アルゴリズムで入力した公開鍵または秘密鍵が持つ情報を指す．
- 復号文の指定
  復号文が持つ情報が何であるかを入力する．

1.3.4　動作確認機能

fCryptica は鍵生成アルゴリズム，暗号化アルゴリズム，復号アルゴリズムをそれぞれ動作確認できる機能を備えている．ここで注意したいのが，鍵生成アルゴリズムは単体で動作確認を行うことができるが，暗号化アルゴリズムは前提となる鍵生成アルゴリズムが記述されていることが必要となり，復号アルゴリズムは鍵生成アルゴリズムおよび暗号化アルゴリズムが記述されている必要がある．

例として，鍵生成アルゴリズムの動作確認の様子を Figure1.5 に示す．

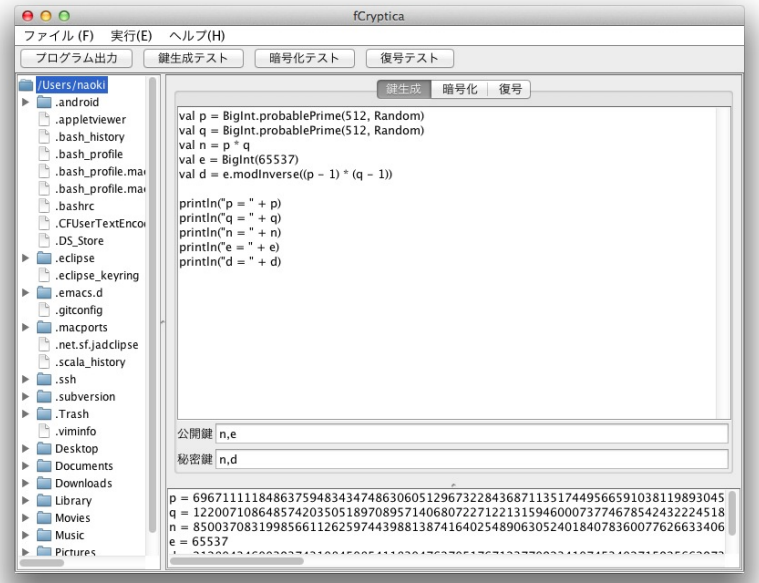

Figure 1.5: fCryptica 鍵生成アルゴリズムの動作確認の様子

例は RSA 暗号の鍵生成アルゴリズムを記述したものである．公開鍵と秘密鍵を指定した状態で,「鍵生成テスト」ボタンまたは「実行→鍵生成テスト」とすると，公開鍵と秘密鍵が出力される，また，鍵生成アルゴリズムの記述の途中に，print 関数を用いて途中の変数の値を出力することもできる．同様に，暗号化テストでは鍵生成テストに加え平文と暗号文が自動出力され，復号テストでは暗号化テストに加え復号文が自動出力される．

これらの出力を見比べることで，ユーザは正常に動作しているかどうかを判断することができる．

1.3.5　プログラム出力機能

fCryptica はユーザが記述した鍵生成アルゴリズム，暗号化アルゴリズムおよび復号アルゴリズムを一つのプログラムとして出力する機能を備えている．出力の際には，記述した暗号アルゴリズムの名前の入力が求められるが，この名前をそのままクラス名として扱う．出力されるプログラムは Figure1.6 のようなクラスで構成される．

この機能により，暗号アルゴリズム以外の余計な記述をすることがなく暗号アルゴリズムの動作確認を簡単に行うことができ，最終的にプログラムとして出力することが可能である．

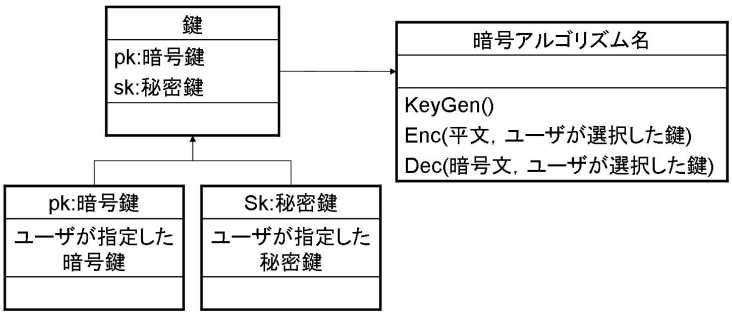

Figure 1.6: fCryptica によって出力されるプログラムのクラス関係

1.3.6 対応数値計算

現在のバージョンである，fCryptica1.0.0 において対応している数値計算は次の通りである．

- 多倍長整数演算
- 複素数演算
- 有理数演算
- 多項式演算
- ベクトル演算
- 行列演算

多倍長整数は Scala 標準ライブラリの多倍長整数型 BigInt を利用しているため，詳しくは Scala の仕様を参照して頂きたい．

1.3.6.1 複素数

複素数型として Complex を実装してある．複素数の実部と虚部に与えられる値は，整数 (多倍長整数を含む) のみとしている．複素数の演算機能を Table1.4 に示す．

Table 1.4: 複素数演算

複素数演算	記述	内容
複素数の宣言	`A = Complex(5,3)`	$A = 5 - 3i$
実部の取得	`A.re`	5
虚部の取得	`A.im`	3
和	`A + A`	$A + A = 10 - 6i$
差	`A - A`	$A - A = 0$
積	`A * A`	$A * A = 16 - 30i$
係数の剰余	`A mod 2`	$A \pmod 2 = 1 + i$

1.3.6.2 有理数

有理数型として Rational を実装してある．有理数の分子と分母に与えられる値は，整数 (多倍長整数を含む) のみとしている．有理数の演算機能を Table1.5 に示す．

Table 1.5: 有理数演算

有理数演算	記述	内容
有理数の宣言	`A = Rational(1,3)`	$A = \frac{1}{3}$
分子の取得	`A.numer`	1
分母の取得	`A.denom`	3
和	`A + A`	$A + A = \frac{2}{3}$
差	`A - A`	$A - A = 0$
積	`A * A`	$A * A = \frac{1}{9}$
商	`A / A`	$A/A = 1$

1.3.6.3 多項式

多項式型として Polynomial を実装してある．係数と指数には整数を，変数には任意の文字を与えることができる．多項式の演算機能を Table1.6,1.7,1.8 に示す．

Table 1.6: 多項式演算

多項式演算	記述	内容
多項式の宣言	`val A = Polynomial("x^2+y+3")` `val B = Polynomial("x+y")`	$A = x^2 + y + 3$ $B = x + y$
和	`A + B`	$A + B = x^2 + x + 2y + 3$
差	`A - B`	$A - B = x^2 - x + 3$
積	`A * B`	$A * B = x^3 + x^2y + xy + 3x + y^2 + 3y$
冪乗	`B^2`	$B^2 = x^2 + 2xy + y^2$
係数の剰余	`A mod 2`	$A \pmod 2 = x^2 + y + 1$

また，変数を一つ指定することでその変数に関する様々な情報が得られる．除算ではどの変数についての除算かを指定する必要がある．上記で定義した `val A = Polynomial("x^2+y+3")` と `val B = Polynomial("x+y")` を例として示す．

Table 1.7: その場で特定の変数を指定する場合の多項式演算

多項式演算	記述	内容
商 (x についての例)	`A / (B,'x)`	$A/B = x - y$
剰余 (x についての例)	`A mod (B,'x)`	$A \pmod B = y^2 + y + 3$
次数の取得 (x についての例)	`A.getDegree('x)`	2
変数の置換 (x についての例)	`A.set('x, 5)` `A.set('x, 'y)`	$y + 28$ $y^2 + y + 3$

Table 1.8: 事前に特定の変数を指定する場合の多項式演算

多項式演算	記述	内容
変数の指定	`Polynomial.init('x)`	x についての多項式
商	`A / B`	$A/B = x - y$
剰余	`A mod B`	$A \pmod B = y^2 + y + 3$
次数の取得	`A.getDegree()`	2
変数の置換	`A.set(5)` `A.set('y)`	$y + 28$ $y^2 + y + 3$

1.3.6.4 ベクトル

ベクトル型として Vektor を実装してある．ベクトルは整数 (多倍長整数を含む)，複素数，有理数，多項式を要素として持つことができる．ベクトルの機能を Table1.9 に示す．

Table 1.9: ベクトル演算

ベクトル演算	記述	内容
ベクトルの宣言	`val A = Vektor(1,2,3)`	$A = (1, 2, 3)$
和	`A + A`	$A + A = (2, 4, 6)$
差	`A - A`	$A - A = (0, 0, 0)$
内積	`A * A`	$A * A = 14$
各要素の積	`A * 2`	$A * 2 = (2, 4, 6)$
各要素の冪乗	`A ^ 2`	$A^2 = (1, 4, 9)$
各要素の商	`A / 2`	$A/2 = (0, 1, 1)$
各要素の剰余	`A % 2 or A mod 2`	$A \pmod 2 = (1, 0, 1)$

上記で定義した `val A = Vektor(1,2,3)` を例として，その他の機能を Table1.10 に示す．

Table 1.10: その他の機能

機能	記述	内容
要素の有無	`A.isEmpty()`	True
要素数の取得	`A.size()`	3
条件付き要素数の取得	`A.count(2<=)`	2
要素の追加 (先頭)	`A.addHead(5)`	$(5, 1, 2, 3)$
要素の追加 (末尾)	`A.addLast(5)`	$(1, 2, 3, 5)$
要素の取得 (2 番目の要素の例)	`A(2)`	2
要素の変更 (2 番目の要素の例)	`A(2)=5`	$(1, 5, 3)$
末尾を除く部分ベクトル	`A.init()`	$(1, 2)$
先頭を除く部分ベクトル	`A.tail()`	$(2, 3)$
先頭 n 個の部分ベクトル (n=2 の例)	`A.take(2)`	$(1, 2)$
先頭 n 個を除く部分ベクトル (n=2 の例)	`A.drop(2)`	(3)
末尾 n 個の部分ベクトル (n=2 の例)	`A.takeRight(2)`	$(2, 3)$
末尾 n 個を除く部分ベクトル (n=2 の例)	`A.dropRight(2)`	(1)
ベクトルの多項式表現	`A.toPolynomial('x)`	$x^2 + 2x + 3$

1.3.6.5 行列

行列型として Matrix を実装してある．行列は整数 (多倍長整数を含む)，複素数，有理数，多項式を要素として持つことができる．行列の機能を Table1.11 に示す．

Table 1.11: 行列演算

行列演算	記述	内容
行列の宣言	`val A = Matrix(Vektor(1,2),Vektor(3,4))`	$A = ((1,2)(3,4))$
和	`A + A`	$A + A = ((2,4)(6,8))$
差	`A - A`	$A - A = ((0,0)(0,0))$
積	`A * A`	$A * A = ((7,10)(15,22))$
冪乗	`A ^ 2`	$A^2 = ((7,10)(15,22))$
転置行列	`A.transpose()`	$A^T = ((1,3)(2,4))$
行列式	`A.determinant()`	-2
逆行列	`A.inverse()`	$A^{-1} = ((-2,1)(\frac{3}{2},-\frac{1}{2}))$
各要素の積	`A * 2`	$A * 2 = ((2,4)(6,8))$
各要素の商	`A / 2`	$A/2 = ((0,1)(0,2))$
各要素の剰余	`A % 2 or A mod 2`	$A \pmod 2 = ((1,0)(1,0))$

上記で定義した `val A = Matrix(Vektor(1,2),Vektor(3,4))` を例として，その他の機能を Table1.12 に示す．

Table 1.12: その他の機能

機能	記述	内容
要素の有無	`A.isEmpty()`	True
要素数の取得	`A.size()`	4
条件付き要素数の取得 (2 以上の値の要素数の例)	`A.count(2<=)`	3
行数の取得	`A.nrow()`	2
列数の取得	`A.ncol()`	2
要素の取得 (1 行 1 列目の例)	`A(1,1)`	1
要素の変更 (1 行 1 列目の例)	`A(1,1)=5`	$((5,2),(3,4))$
行ベクトルの取得 (2 行目の例)	`A.row(2)`	$(3,4)$
列ベクトルの取得 (2 列目の例)	`A.col(2)`	$(2,4)$

付録 A

OpenSSL による RSA 暗号の実装

```
#include <stdio.h>
#include <string.h>
#include <openssl/bn.h>
#include <openssl/engine.h>

#define KEYBIT_LEN  128

typedef struct {
  Publickey pubkey;
  Privatekey prikey;
} KeyPair;

typedef struct {
  BIGNUM n;
  BIGNUM e;
} Publickey;

typedef struct {
  BIGNUM n;
  BIGNUM d;
} Privatekey;

KeyPair KeyGen(){
  BIGNUM *bn_e = BN_new();
  BN_set_word(bn_e, 65537);
  BIGNUM *bn_n = BN_new();
  BIGNUM *bn_d = BN_new();
  BN_CTX *bn_ctx_tmp = BN_CTX_new();

  BIGNUM *bn_p = BN_new();        /* 素数 p */
  BIGNUM *bn_q = BN_new();        /* 素数 q */
  BN_generate_prime(bn_p, KEYBIT_LEN/2, 1, NULL, NULL, NULL, NULL);
  BN_generate_prime(bn_q, KEYBIT_LEN/2, 1, NULL, NULL, NULL, NULL);

  BIGNUM *bn_pm = BN_new();       /* p-1 (p minus 1) */
  BIGNUM *bn_qm = BN_new();       /* q-1 (q minus 1) */
```

```
  BIGNUM *bn_pmqm = BN_new();     /* (p-1)*(q-1) */

  BN_mul(bn_n, bn_p, bn_q, bn_ctx_tmp);
  BN_sub(bn_pm, bn_p, bn_one);
  BN_sub(bn_qm, bn_q, bn_one);
  BN_mul(bn_pmqm, bn_pm, bn_qm, bn_ctx_tmp);
  BN_mod_inverse(bn_d, bn_e, bn_pmqm, bn_n, bn_ctx_tmp);

  BN_free(bn_p); BN_free(bn_q); BN_free(bn_pm); BN_free(bn_qm);
  BN_free(bn_pmqm); BN_free(bn_gcd_pmqm);

  Publickey pubkey = {bn_n, bn_e}
  Privatekey prikey = {bn_n, bn_d}

  KeyPair keypair = {pubkey, prikey}
  return key pair;
}

// prain は BIGNUM 型の平文
BIGNUM Enc(pubkey, prain){
  BIGNUM *bn_encrypt = BN_new();
  BN_CTX *bn_ctx_tmp = BN_CTX_new();
  BN_mod_exp(bn_encrypt, prain, pubkey.e, pubkey.n, bn_ctx_tmp);
  return bn_encrypt;
}

BIGNUM Dec(prikey, bn_encrypt){
  BIGNUM *bn_decrypt = BN_new();
  BN_CTX *bn_ctx_tmp = BN_CTX_new();
  BN_mod_exp(bn_decrypt, bn_encrypt, prikey.d, prikey.n, bn_ctx_tmp);
  return bn_encrypt;
}
```

付録B

Java 標準ライブラリによる RSA 暗号の実装

```
import java.math.BigInteger;
import java.security.SecureRandom;
import java.util.*;

public class RSA {
  /* 鍵生成 */
  KeyPair KeyGen(){
    SecureRandom random = new java.security.SecureRandom();

    BigInteger e = new BigInteger("65537");
    BigInteger p = BigInteger.probablePrime(64, random);
    BigInteger q = BigInteger.probablePrime(64, random);

    BigInteger n = p.multiply(q);

    BigInteger pm = p.subtract(BigInteger.ONE);
    BigInteger qm = q.subtract(BigInteger.ONE);
    BigInteger pmqm = pm.multiply(qm);
    BigInteger d = e.modInverse(pmqm)

    PublicKey pubkey = new PublicKey(n, e);
    PrivateKey prikey = new PublicKey(n, d);

    KeyPair keypair = new KeyPair(pubkey, prikey);
    return keypair;
  }

  /* 暗号化 */
  BigInteger Enc(pubkey, plain){
    BigInteger encrypto = plain.modpow(pubkey.e, pubkey.n);
    return encrypto;
  }

  /* 復号 */
  BigInteger Dec(prikey, encrypto){
    BigInteger decrypto = encrypto.modpow(prikey.d, prkey.n);
```

```
    return decrypto;
  }
}
```

```
public class KeyPair{
  public PublicKey pubkey;
  public PrivateKey prikey;
  KeyPair(PublicKey pubkey, PrivateKey prikey){
    this.pubkey = pubkey;
    this.prikey = prikey;
  }
}
```

```
import java.math.BigInteger;
public class PublicKey{
  public BigInteger n;
  public BigInteger e;
  PublicKey(BigInteger n, BigInteger e){
    this.n = n;
    this.e = e;
  }
}
```

```
import java.math.BigInteger;
public class PrivateKey{
  public BigInteger n;
  public BigInteger d;
  PrivateKey(BigInteger n, BigInteger d){
    this.n = n;
    this.d = d;
  }
}
```

付録 C

fCryptica による RSA 暗号の実装

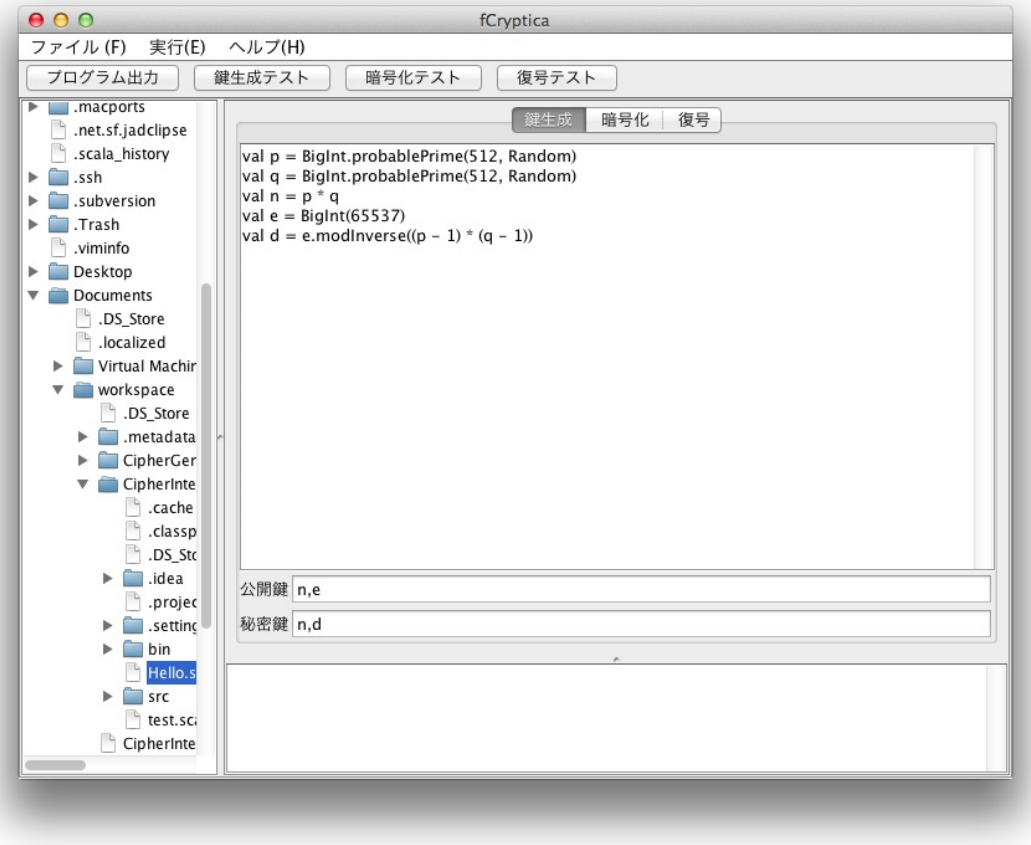

Figure 1.7: 鍵生成アルゴリズム入力画面

実際の記述

```
val p = BigInt.probablePrime(512, Random)
val q = BigInt.probablePrime(512, Random)
val n = p * q
val e = BigInt(65537)
val d = e.modInverse((p - a) * (q - 1))
```

公開鍵： n,e

秘密鍵： n,d

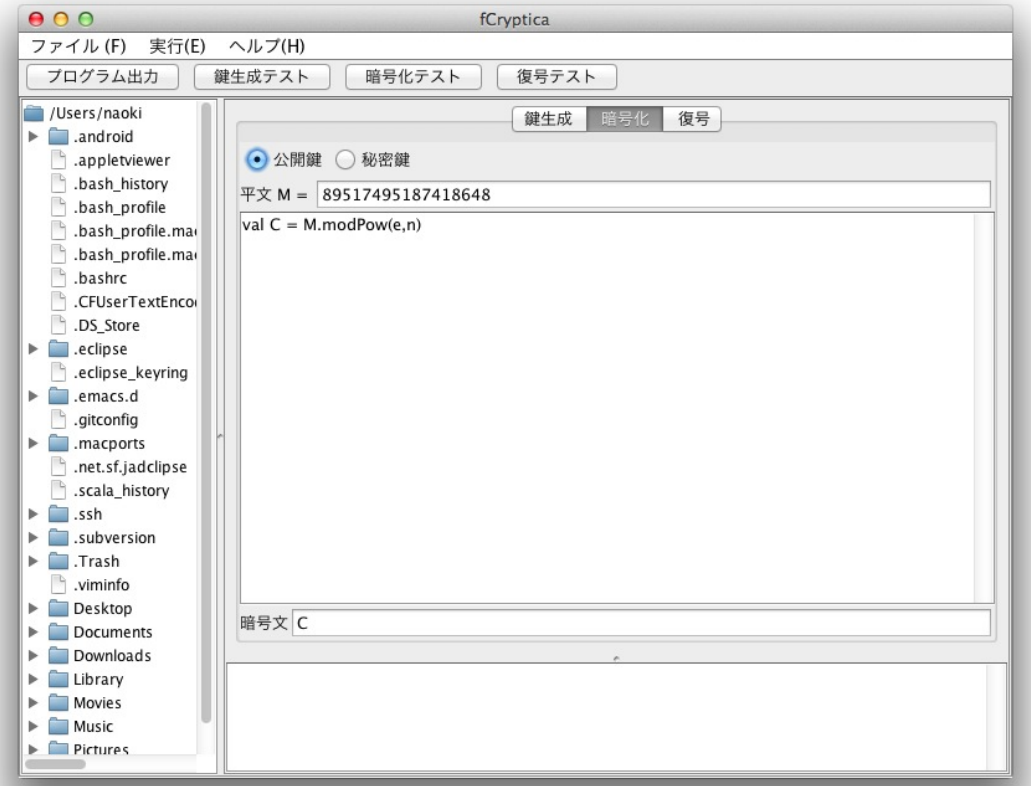

Figure 1.8: 暗号化アルゴリズム入力画面

実際の記述

```
val C = M.modPow(e,n)
```

暗号文： C

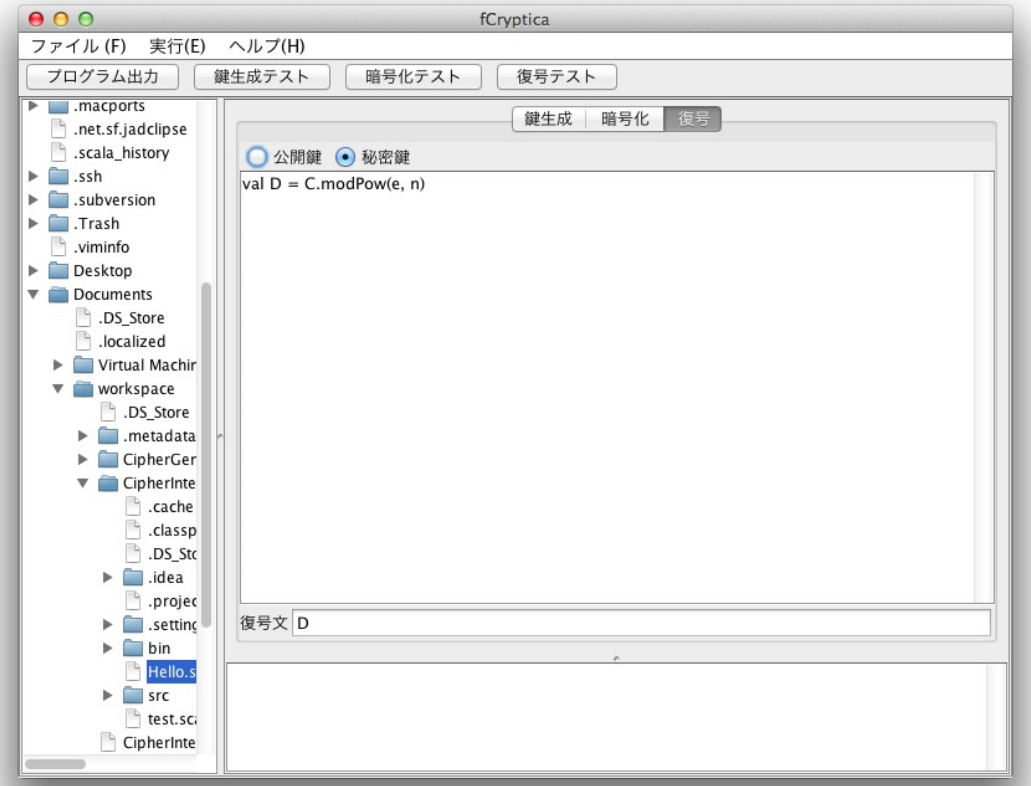

Figure 1.9: 復号アルゴリズム入力画面

実際の記述

```
val D = C.modPow(d,n)
```

復号文：D