Universität Bremen

Praktische Informatik 3 WS 20/21

Christoph Lüth
Thomas Barkowsky
Tobias Brandt
Alexander Krug
Tarek Soliman
Robert Sachtleben

3. Übungsblatt

Ausgabe: 23.11.2020
Abgabe: 30.11.2020 12:00

3.1 *Konversionen* *3 Punkte*

Wir fangen einfach an und implementieren zwei Funktionen, die eine Zahl in die Darstellung zu einer gegeben Basis konvertieren bzw. wieder zurück konvertieren:

```
convertToBase :: Integer→ Integer→ [Integer]
convertFromBase :: Integer→ [Integer]→ Integer
```

In der Liste ist die geringste Stelle an vorderster Stelle, das macht die Konversion einfacher. Beispielsweise ist

```
convertToBase 10 1984 ⇝ [4, 8, 9, 1]
convertToBase 8 1984 ⇝ [0, 0, 7, 3]
convertFromBase 8 [0, 0, 7, 3] ⇝ 1984
convertFromBase 2 [1, 1, 0, 1] ⇝ 11
```

Die Basis ist hier immer eine positive Zahl $b > 1$. Achtung, Zahlen (`Integer`) werden in der Kommandozeile immer zur Basis 10 ausgegeben, was aber noch keine Darstellung zur Basis 10 im Sinne dieser Aufgabe ist.

Mit diesen Funktionen werden jetzt zwei Funktionen implementiert, die eine Zeichenkette als eine Zahl zur Basis 256 auffassen:

```
stringAsNumber :: String→ Integer
numberAsString :: Integer→ String

stringAsNumber "foo" ⇝ 7303014
numberAsString 7496034 ⇝ "bar"
```

Hinweis: Für den zweiten Teil müssen wir einzelne Zeichen (`Char`) in `Integer` und zurück konvertieren. Dabei helfen uns folgende vordefinierte Funktionen:

```
ord :: Char→ Int -- Aus Data.Char
chr :: Int→ Char -- Aus Data.Char
fromInteger :: Integer→ Int
toInteger :: Int→ Integer
```

Die Funktionen `convertFromBase` und `stringAsNumber` können Null-Werte am Ende nicht bewahren, d.h.

```
convertFromBase 2 [0, 1, 0, 0] ⇝ 2
stringAsNumber "A\NUL" ⇝ 65
```

3.2 *Asymmetrische Verschlüsselung* *7 Punkte*

Bei einem asymmetrischen Schlüsselverfahren hat jeder Teilnehmer X einen öffentlichen Schlüssel P_X, den jeder andere Teilnehmer kennt, und einen geheimen Schlüssel S_X, den nur X kennt. Eine Nachricht M von A ("Alice") an B ("Bob") wird mit dem öffentlichen Schlüssel P_B von B verschlüsselt, $C = P_B(M)$. B kann die Nachricht dann mit seinem geheimen Schlüssel S_B entschlüsseln, wenn folgendes gilt:

$$M = S_B(P_B(M)) \tag{1}$$

Entscheidend für die Sicherheit ist hier, dass aus dem öffentlichen Schlüssel P_X der geheime Schlüssel S_X nicht errechnet werden kann, oder nur mit sehr großem Aufwand.

Das *RSA-System* ist eine Übung in angewandter Algebra. Ohne hier vertieft auf die mathematischen Grundlagen einzugehen: ein *Schlüssel* besteht immer aus zwei Zahlen (k, n) mit $k <= n$, wobei n die Schlüsselgröße ist. Die Verschlüsselungsfunktion für eine Nachricht M und einen öffentlichen Schlüssel $P_X = (e, n)$ ist

$$P_X(M) = M^e \bmod n, \tag{2}$$

und als Entschlüsselungsfunktion ergibt sich für einen privaten Schlüssel $S_X = (d, n)$ und eine Nachricht C,

$$S_X(C) = C^d \bmod n. \tag{3}$$

Für geeignete Schlüsselpaare $P_X = (e, n)$ und $S_X = (d, n)$ gilt dann

$$S_X(P_X(M)) = ((M^e) \bmod n)^d \bmod n = M^{ed} \bmod n = M$$

1. Wir implementieren zuerst die Kernfunktionalität.

   Für die Verschlüsselung ist es wichtig, den Ausdruck M^e in (2) nicht direkt zu berechnen (die Zahl wäre viel zu groß). Deshalb implementieren wir eine schnelle Version der Exponentation modulo n:

   ```
   modexp :: Integer→ Integer→ Integer→ Integer
   ```

   Diese reduziert Exponentation auf Quadrierung und nutzt die Tatsache, dass $a^{2n} = (a^n)^2$, d.h.

   $$a^n \bmod p = \begin{cases} (a^{n \div 2} \bmod p)^2 \bmod p & n \text{ gerade} \\ (a(a^{n \div 2} \bmod p)^2) \bmod p & n \text{ ungerade} \end{cases}$$

   wobei $a^0 \bmod p = 1$. Sie können diese rekursive Formulierung direkt in Haskell übersetzen. Alles andere ist für Zahlen in der Größe, mit der wir hier rechnen, zu ineffizient — probieren Sie es aus!

   Die Verschlüsselung benutzt als Schlüssel Paare aus beliebig großen, ganzen Zahlen, und besteht aus zwei Funktionen:

   ```
   data Key = Key { factor :: Integer, size :: Integer }

   encrypt :: Key→ Integer→ Integer
   decrypt :: Key→ Integer→ Integer
   ```

   die eine Nachricht (eine Zahl m) mit einem öffentlichen Schlüssel (e, n) verschlüsselt bzw. einem privaten Schlüssel (c, n) entschlüsselt. Dabei muss gelten, dass $e < n, c < n$. In der Vorlage sind geeignete Schlüsselpaare zu finden, bspw. `prk64` und `pbk64`, so dass

   ```
   encrypt prk64 1234567890  ⇝  6020575960829543985
   decrypt pbk64 6020575960829543985  ⇝  1234567890
   ```

   3 Punkte

2. Zeichenketten, die erste.

   Nun ist die Verschlüsselung von ganzen Zahlen, auch wenn diese sehr groß werden können (bei den typischerweise verwendeten Schlüsseln mehr als 30 Stellen), auf die Dauer nur für Zahlenfetischisten spannend. Wir wollen natürlich Textdateien, Bilder oder zumindest Zeichenketten verschlüsseln.

   In einem ersten Schritt wandeln wir dazu jedes Zeichen in eine Zahl um, und verschlüsseln diese. Damit ergibt sich als verschlüsselter Text eine Liste von (sehr großen) ganzen Zahlen:

   ```
   simpleEncrypt :: Key → String→ [Integer]
   simpleDecrypt :: Key→ [Integer]→ String

   simpleDecrypt (pbk128 (simpleEncrypt prk128 "Foobaz?"))  ⇝  "Foobaz?"
   ```

   1 Punkte

3. Zeichenketten, die zweite:

   Der Nachteil der einfachen Methode ist, dass sich die Größe der Nachrichten enorm vergrößert, da jedes Zeichen in einen Zahl zwischen 1 und der (sehr großen) Schlüsselgröße n abgebildet wird. Darüber hinaus ist diese Verschlüsselung leicht anzugreifen, da jeder Buchstabe auf die gleiche, große Zahl abgebildet wird — man braucht also nur längere Texte, und kann dann aus der Häufigkeit des Auftretens schließen, welche Zahl welches Zeichen kodiert.

   In einem zweiten Schritt unterteilen wir deshalb die Nachricht in Blöcke zu k Zeichen, wobei ein Block von k Zeichen eine Zahl zu der Basis 256 darstellt. Die Länge k der Teilstücke muss so gewählt werden, dass die damit zu der Basis 256 dargestelltem Zahlen immer kleiner als die Größe des Schlüssels n sind. Also wählen wir als Länge k die um eins verringerte Länge der Darstellung des Schlüssels n zur Basis 256 (`convertToBase` aus Aufgabe 3.1). Das berechnet die Funktion `chunkLength`:

   ```
   chunkLength :: Key→ Integer

   chunkLength prk1024 ⇝ 127
   ```

   Eine Nachricht wird jetzt wie folgt verschlüsselt:

   (a) die zu verschlüsselnde Botschaft wird in Blöcke der berechneten Länge aufgeteilt;
   (b) jeder dieser Zeichenketten wird mittels `stringAsNumber` (aus Aufgabe 3.1) in einen `Integer` konvertiert;
   (c) jede dieser Zahlen wird mit `encrypt` verschlüsselt.

   Die Entschlüsselung läuft entsprechend umgekehrt:

   (a) in der Liste von `Integer` wird jeder mit `decrypt` entschlüsselt, und mittels `numberAsString` zu einer Zeichenkette konvertiert;
   (b) die Listen von Zeichenketten werden zu einer einzigen Zeichenkette zusammengefügt.

   Implementieren Sie zwei Funktionen

   ```
   encryptMsg :: Key→ String→ [Integer]
   decryptMsg :: Key→ [Integer]→ String
   ```

   welche diese Funktionalität bereitstellen.

   In der Vorlage ist eine verschlüsselte Nachricht `secret` hinterlegt. Wenn alles funktioniert, können Sie mit

   ```
   putStrLn (decryptMsg pbk2048 secret)
   ```

   das Geheimnis entschlüsseln.

   3 Punkte

Hinweise:

1. Um einen String (eine Liste) in Teilstücke (*chunks*) gegebener Länge aufzuspalten können wir die Funktion `chunksOf` aus dem Modul `Data.List.Split` nutzen (wird in der Vorlage importiert):

   ```
   chunksOf :: Int→ [α]→ [[α]]
   ```

   Weiterhin verkettet folgende Funktion eine Liste von Listen zu einer Liste:

   ```
   concat :: [[α]]→ [α]
   ```

2. Die Vorlage enthält 5 vordefinierte Schlüsselpaare von verschiedener Größe, von 64 bis zu 2048 Bit. Sie können diese zu Testzwecken nutzen.

   Die Vorlage implementiert Unit-Tests mit einem längeren String, diese Schlüssel nutzen. Wichtig ist, dass folgende Gleichung gilt, wenn `k1` und `k2` ein gültiges Schlüsselpaar sind:

   $$\texttt{decryptMsg k1 (encryptMsg k2) m} == \texttt{m} \tag{4}$$

   Aufgrund der Eigenschaften von `stringAsNumber` oben gilt diese Gleichung nur für Zeichenketten `m` ohne Null-Werte am Ende.[1]

[1] Um das Verfahren robust zu machen könnte die Verschlüsselung immer ein letztes Nicht-Null-Zeichen anfügen, welches die Verschlüsselung wieder entfernt, aber da Null-Werte sowieso Strings beenden ist das kein weiteres Problem.