Lista zagadnień nr 3

Przed zajęciami

W tym tygodniu zajmować się będziemy przede wszystkim podstawami abstrakcji danych i programowaniem z użyciem list, pojawi się też pierwsze z bardziej rozbudowanych zadań domowych. Przed zajęciami należy przeczytać ze zrozumieniem **Rozdziały 2.1** i **2.2.1** podręcznika, należy znać pojęcia **konstruktorów**, **selektorów** i **predykatów**, a także potrafić opowiedzieć o łączących je **równaniach** na przykładach **par**, **list** i wybranego abstrakcyjnego przykładu. Należy też przeczytać treść zadania na pracownię, żeby móc wyjaśnić ewentualne wątpliwości w czasie ćwiczeń.

Ćwiczenie 1.

Na wykładzie przyjęliśmy reprezentację liczb wymiernych jako par składających się z licznika i mianownika (gdzie licznik i mianownik są liczbami całkowitymi, i mianownik jest różny od zera). Alternatywnie możemy wybrać reprezentację jako *list dwuelementowych* (tj. list których długość jest równa 2) — która może być nieco ładniej wyświetlana przez interpreter.

Zdefiniuj konstruktor `make-rat`, selektory `rat-num` i `rat-den`, i predykat `rat?` dla tej alternatywnej reprezentacji, a następnie pokaż że oczekiwane równania będą zachodzić również przy tej reprezentacji.

Ćwiczenie 2.

Wektory zaczepione na płaszczyźnie możemy reprezentować jako pary punktów: odpowiednio początek i koniec. Zdefiniuj konstruktor `make-vect`, predykat `vect?` i selektory `vect-begin` i `vect-end`, definiujące reprezentację wektora zaczepionego za pomocą punktów. Punkt możemy zdefiniować podobnie, jako parę współrzędnych kartezjańskich. Podobnie jak w przypadku wektorów, zdefiniuj konstruktor `make-point`, predykat `point?` i selektory `point-x` i `point-y`.

Następnie, dbając o zachowanie abstrakcji danych, zdefiniuj następujące procedury:

- `vect-length`, obliczającą długość wektora,
- `vect-scale`, taką że `(vect-scale v k)` znajduje wektor o początku w punkcie `(vect-begin v)`, ale który jest `k`-krotnie dłuższy,
- `vect-translate`, taką że `(vect-translate v p)` znajduje wektor o początku w punkcie `p` który jest przesunięciem wektora `v`.

Mogą Ci się przydać następujące procedury:

```
(define (display-point p)
  (display "(")
  (display (point-x p))
  (display ", ")
  (display (point-y p))
  (display ")"))

(define (display-vect v)
  (display "[")
  (display-point (vect-begin v))
  (display ", ")
  (display-point (vect-end v))
  (display "]"))
```

Na zajęciach

Ćwiczenie 3.

Wektory zaczepione możemy reprezentować również jako trójki składające się z punktu początkowego, kierunku (kąta z przedziału $[0, 2\pi)$) i długości. Zaimplementuj taką reprezentację wektorów i zdefiniuj dla niej procedury z poprzedniego zadania.

Ćwiczenie 4.

Zdefiniuj jednoargumentową procedurę `reverse`, która przyjmuje listę i zwraca listę elementów w odwrotnej kolejności. Podaj dwie implementacje, rekurencyjną i iteracyjną, i udowodnij ich równoważność. Jaką złożoność mają obie implementacje?

Ćwiczenie 5.

Zdefiniuj procedurę `insert`, taką że dla posortowanej (rosnąco) listy liczb `xs` wywołanie `(insert xs n)` obliczy posortowaną listę w której `n` zostanie wstawione na właściwe miejsce. Wykorzystaj procedurę `insert` do zaimplementowania algorytmu sortowania przez wstawianie.

Ćwiczenie 6.

Permutacją ciągu $x_1, x_2, \ldots, x_n$ nazywamy dowolny ciąg $x_{i_1}, x_{i_2}, \ldots, x_{i_n}$, gdzie $i_1, \ldots i_n$ są parami różnymi liczbami z przedziału $1, \ldots, n$. Permutacje powstają zatem przez przestawienie kolejności elementów na liście.

Zdefiniuj procedurę `permi` zwracającą listę wszystkich permutacji danej listy, działającą zgodnie z poniższym schematem:

- Jedyną permutacją listy pustej jest lista pusta.
- Aby wygenerować permutację niepustej listy wygeneruj permutację jej ogona i wstaw jej głowę w dowolne miejsce uzyskanej permutacji.

Ćwiczenie 7.

Udowodnij że `append` wraz z listą pustą `null` tworzą monoid:

- Dla dowolnych `xs`, `ys` i `zs`, jeśli `(list? xs)` i `(list? ys)` to `(append (append xs ys) zs)` $\equiv$ `(append xs (append ys zs))`
- Dla dowolnego `xs`, jeśli `(list? xs)` to `(append xs null)` $\equiv$ `xs`
- Dla dowolnego `xs`, `(append null xs)` $\equiv$ `xs`.

Ćwiczenie 8.

Niech `xs` i `ys` będą dowolnymi listami (tj. `(list? xs)` i `(list? ys)` są prawdą). Czemu jest wtedy równe:

- `(reverse (append xs ys))`,
- `(map f (append xs ys))` (dla pewnej procedury jednoargumentowej `f`),
- `(filter p? (append xs ys)` (dla pewnego predykatu jednoargumentowego `p?`)?

Udowodnij odpowiednie twierdzenia.

Ćwiczenie 9.

Uogólnij procedurę `append` z wykładu tak żeby mogła przyjmować dowolnie dużo list jako argumenty (obliczenie `(append)` powinno dawać w wyniku listę pustą).

Zadanie domowe (na pracownię)

Szyfry i tajne kody były wykorzystywane przez ludzkość od tysiącleci. Tym bardziej zaskakujące że istotny przełom w technologii szyfrowania wiadomości dokonał się niewiele ponad 40 lat temu. Istotnym problemem w praktyce

szyfrowania była zawsze konieczność przechowywania i przekazywania tajnego kodu, ustalonego przez obie komunikujące się strony. Kluczowa obserwacja, na której oparta jest istotna część współczesnej kryptografii, mówi że istnieją kryptosystemy w których znajomość klucza użytego do *zakodowania* wiadomości nie jest pomocna przy jej *dekodowaniu*: ta obserwacja prowadzi do *kryptografii z kluczem publicznym*.[1]

W systemie w którym znajomość klucza *kodującego* nie pomaga w *dekodowaniu* wiadomości najtrudniejsza logistycznie część działającego kryptosystemu — ustalenie wspólnego, tajnego hasła — przestaje być problemem: cały świat może znać hasło służące do kodowania wiadomości jednej ze stron (i wysyłać jej zaszyfrowane wiadomości), ale nikt poza osobą zainteresowaną nie będzie ich w stanie odczytać. Jednym z pierwszych praktycznych kryptosystemów opartych na idei klucza publicznego było RSA, opracowane przez naukowców z MIT: Rivesta, Shamira i Adelmana,[2] i uproszczoną wersją tego kryptosystemu zajmiemy się na pracowni.

RSA — teoria i implementacja

RSA reprezentuje grupy znaków z szyfrowanego tekstu jako liczby naturalne i używa specjalnej transformacji na liczbach aby zaszyfrować tekst. Podstawą schematu RSA jest wybór dwóch dużych liczb pierwszych, p i q. Mając takie liczby definiujemy $n = pq$ i $m = (p-1)(q-1)$, a także wybieramy e takie że e jest względnie pierwsze z m. Wybór tych liczb daje nam *klucz publiczny* służący do szyfrowania wiadomości, który stanowi para (n, e). Transformacja szyfrująca liczbę s (reprezentującą pewną grupę znaków) jest dana jako:

$$\bar{s} = s^e \mod n.$$

Analogiczną operację, używającą specjalnej liczby d wykonujemy żeby odszyfrować tekst: bierzemy

$$s' = \bar{s}^d \mod n,$$

przy czym liczba d jest wybrana tak, żeby $s = s'$ dla dowolnej wiadomości s, tj. że

$$s = (s^e)^d \mod n.$$

Można pokazać, że taką własność będzie miała liczba d taka że

$$de = 1 \mod m,$$

[1] Obserwacja pochodzi od Diffie'ego i Hellmana ze Stanford University, choć wcześniej dokonali jej również matematycy brytyjskiego wywiadu — jednak badania te pozostały utajnione do drugiej połowy lat 90-tych.

[2] Jak poprzednio, brytyjski wywiad był szybszy, ale wyniki długo pozostały tajne.

okazuje się też, że znając e i m, możemy łatwo policzyć d.

Podsumowując, żeby wygenerować parę kluczy RSA (prywatny i publiczny) wybieramy duże liczby pierwsze p i q, obliczamy $n = pq$, wybieramy e i używamy go żeby policzyć d. Parę (n, e) ogłaszamy jako swój klucz publiczny, używany do szyfrowania wiadomości do nas, zaś parę (n, d) zachowujemy w tajemnicy jako klucz prywatny, służący do odszyfrowywania wiadomości. Bezpieczeństwo systemu RSA opiera się na fakcie że nie znamy szybkiej metody wyznaczenia d na podstawie pary (n, e): w ogólnym przypadku najlepsze co potrafimy zrobić to rozłożyć n na czynniki pierwsze — co jest bardzo trudne i czasochłonne gdy n jest iloczynem dwóch dużych liczb pierwszych.

Podpisy cyfrowe Okazuje się że kryptografia z kluczem publicznym może służyć do rozwiązania innego istotnego problemu bezpiecznej komunikacji cyfrowej: ochrony przed fałszerstwem wiadomości. Jeśli każdy zna nasz klucz publiczny, fałszerz może wysłać nam zaszyfrowany komunikat podszywając się pod kogoś innego. Jak teraz rozpoznać takie fałszerstwo?

Rozwiązanie zaproponowane przez Diffie'ego i Hellmana jest następujące: należy wziąć tekst wiadomości i zastosować do niego uzgodnioną wcześniej *funkcję haszującą*,[3] która przetwarza tekst w jedną niewielką liczbę (w ogólności wiele tekstów da tę samą liczbę, ale kolizje te będą rzadkie). Tę liczbę transformujemy swoim kluczem *prywatnym*: tak przetworzona staje się podpisem cyfrowym, dołączanym do naszej wiadomości. Każdy kto taką wiadomość otrzyma, może sprawdzić czy nie została sfałszowana aplikując do podpisu nasz klucz *publiczny* i porównując wynik z wynikiem funkcji haszującej zaaplikowanej do tekstu wiadomości. Co więcej, tej metody można użyć również do podpisania *zaszyfrowanego* tekstu, dając nam równocześnie tajność i pewność komunikacji.

Implementacja RSA Implementacja (większości) protokołu RSA znajduje się w dołączonym pliku. Kluczową częścią jest implementacja szybkiego modularnego potęgowania z części 1.2 podręcznika. Kod dostarcza dwóch abstrakcyjnych struktur danych: kluczy (składających się z modułu i wykładnika) i par kluczy (składających się z klucza prywatnego i publicznego). Dla dowolnego klucza definiujemy transformację RSA na pojedynczej liczbie, `RSA-transform`, opisaną powyżej.

Plik z kodem dostarcza też procedurę `generate-RSA-key-pair` generującą pary złożone z klucza prywatnego i publicznego poprzez poszukiwanie losowych liczb pierwszych z ustalonego zakresu i znajdowanie odwrotności

[3] *Hash function*, bywa tłumaczona również jako funkcja skrótu lub funkcja mieszająca

multiplikatywnej dla wybranego wykładnika klucza publicznego. Zakres liczb pierwszych jest specjalnie niewielki, żeby klucze można było łatwo złamać, jednak ogólna zasada działania pozostaje niezmieniona. Testowanie pierwszości oparte jest na teście probabilistycznym z rozdziału 1.2 podręcznika.

Procedury `RSA-encrypt` i `RSA-decrypt` szyfrują i odszyfrowują tekst przy użyciu podanego klucza. Kod jest rozbity na dwie części: konwersję między napisem a listą liczb, i stosowaniem szyfrowania RSA do otrzymanej listy liczb. Transformacja napisów do list liczb nie jest dla nas w tej chwili interesująca — dostarczony kod należy traktować jako bibliotekę. Istotne są za to procedury szyfrujące i deszyfrujące: `RSA-convert-list` i `RSA-unconvert-list` (implementacja tej drugiej jest częścią zadania). Warto zwrócić uwagę, że (wbrew intuicji) nie kodujemy każdej liczby osobno: zamiast tego bezpieczniej jest zakodować pierwszą liczbę, odjąć zakodowaną liczbę od drugiej (modulo n), zakodować wynik — i powtarzać ten schemat. W ten sposób każdy kolejny element zakodowanego ciągu zależy od wszystkich elementów ciągu wejściowego.

Wreszcie, żeby zaimplementować podpisy cyfrowe, potrzebujemy funkcji haszującej. Naiwną implementację dostarcza procedura `compress` (o tym co sprawia że funkcja haszująca jest dobra będziecie się uczyć na AiSD — na potrzeby tego zadania naiwna implementacja wystarczy).

Ćwiczenie 10.

Kod który dostaliście nie jest kompletny: brakuje w nim implementacji trzech istotnych procedur. Waszym zadaniem jest zaimplementowanie ich.

Dekodowanie szyfrogramu Pierwszą z procedur które należy zaimplementować jest `RSA-unconvert-list`, odwracająca transformację wykonywaną przez `RSA-convert-list`. Wskazówka: Ta procedura powinna być bardzo podobna do `RSA-convert-list`; jeśli Twój kod bardzo od niej odbiega, prawdopodobnie robisz coś źle (w takiej sytuacji warto poprosić o pomoc prowadzącego). Przetestuj swoją procedurę używając podanych przykładowych par kluczy.

Generowanie kluczy Proces generowania kluczy jest zaimplementowany niemal w całości. Brakujący fragment to istotna część znajdowania odwrotności multiplikatywnej wykładnika klucza publicznego (a więc znajdowanie na jego podstawie wykładnika klucza prywatnego). Aby znaleźć d takie że $de = 1 \mod m$, wystarczy rozwiązać równanie $km + de = 1$. Twoim zadaniem jest znaleźć rozwiązanie tego drugiego równania znając m i e. Brakująca procedura, `solve-ax+by=1` ma rozwiązywać właśnie takie równania.

Wskazówka: Aby rozwiązać równanie postaci $ax + by = 1$ można zastosować podobny schemat co przy znajdowaniu największego wspólnego podzielnika przy pomocy algorytmu Euklidesa. Załóżmy że $a > b$. Wtedy mamy $a = bq + r$ dla pewnych q i r (takich że $r < b$). Jeśli teraz rozwiążemy równanie $bx' + ry' = 1$, to łatwo możemy obliczyć wartości x i y.

Podpisywanie wiadomości Ostatnią częścią zadania na ten tydzień jest implementacja protokołu szyfrowania i podpisywania wiadomości opisanego powyżej. Zacznij od zdefiniowania (prościutkiej) struktury danych przechowującej podpisane wiadomości, o konstruktorze `make-signed-message` i selektorach `signature` i `message`.

Następnie zdefiniuj procedurę `encrypt-and-sign`, która przyjmuje jako argumenty tekst wiadomości, klucz publiczny adresata i klucz prywatny nadawcy. Procedura powinna zaszyfrować wiadomość używając klucza adresata, obliczyć cyfrowy podpis na podstawie zaszyfrowanego tekstu, i połączyć te dwie części w zaszyfrowaną i podpisaną wiadomość.

Zdefiniuj też procedurę odwrotną, `authenticate-and-decrypt`, która przyjmuje podpisaną i zaszyfrowaną wiadomość, klucz prywatny adresata i klucz publiczny nadawcy. Jeśli podpis cyfrowy jest zgodny z oczekiwanym, procedura powinna zwrócić odszyfrowany tekst; jeśli nie — powinna zasygnalizować problem.

Rozwiązania, jak do tej pory, należy wysyłać w systemie SKOS do 12 marca 2018, godz 06.00. Plik o nazwie `nazwisko-imie.rkt` powinien być dostarczonym szablonem zmienionym *wyłącznie* w oznaczonych miejscach — należy w definicji czterech powyższych funkcji zastąpić zgłoszenie błędu właściwą implementacją.

Uwaga: Powyższy problem nie wymaga napisania dużej ilości kodu. Wymaga natomiast przejrzenia nietrywialnego programu tak aby zidentyfikować miejsca które trzeba zrozumieć — i te, których zrozumieć *nie trzeba*. Warto przeczytać dostarczony kod zawczasu, poznać jego strukturę i spokojnie wypełnić brakujące miejsca, a nie próbować wszystko zrobić „za pięć dwunasta".