Zadanie 1 (4p.)

Samefringe Problem. Definiujemy typ danych do reprezentacji drzew binarnych przechowujących wartości w liściach:

```
type 'a btree = Leaf of 'a | Node of 'a btree * 'a
btree
```

Dwa drzewa binarne typu `t btree` mają jednakowe brzegi (zakładamy, że obiekty typu `t` są porównywalne), jeśli listy utworzone przez odczytanie wartości w ich liściach od lewej do prawej są równe. Na przykład drzewa

```
Node (Node (Leaf 1, Leaf 2), Leaf 3)
```

i

```
Node (Leaf 1, Node (Leaf 2, Leaf 3))
```

mają jednakowe brzegi, równe `[1; 2; 3]`.

1. Napisz funkcję rozstrzygającą czy dwa drzewa mają jednakowe brzegi, bazując bezpośrednio na definicji i nie dbając o efektywność rozwiązania.
2. Wykorzystując pojęcie odroczonego obliczenia, napisz efektywną i czysto funkcyjną wersję funkcji `samefringe`, tj. taką, która przerywa obliczenia w momencie napotkania pierwszej różnicy między brzegami drzew. Podpowiedź: należy odraczać trawersowanie prawego poddrzewa.

Zadanie 2 (4p.)

Definiujemy typ danych do reprezentacji drzew binarnych przechowujących wartości zarówno w węzłach jak i w liściach:

```
type 'a btree = Leaf of 'a | Node of 'a btree * 'a *
'a btree
```

1. Napisz funkcję numerującą węzły i liście drzewa binarnego w kolejności przechodzenia go w głąb (preorder). Na przykład, tak ponumerowaną wersją drzewa

   ```
   Node (Node (Leaf 'a', 'b', Leaf 'c'), 'd', Leaf
   'e')
   ```

   jest

   ```
   Node (Node (Leaf 3, 2, Leaf 4), 1, Leaf 5).
   ```

2. Napisz funkcję numerującą węzły i liście drzewa binarnego w kolejności

przechodzenia go wszerz. Na przykład, tak ponumerowaną wersją drzewa

```
Node (Node (Leaf 'a', 'b', Leaf 'c'), 'd', Leaf
'e')
```

jest

```
Node (Node (Leaf 4, 2, Leaf 5), 1, Leaf 3).
```

Podpowiedź: lasy numeruje się łatwiej niż drzewa.

Zadanie 3 (4p.)

Tablica funkcyjna to struktura danych, która podobnie jak tablica imperatywna, pozwala na swobodny dostęp do swoich składowych (poprzez ich indeksy w tablicy). Jednakże, w przeciwieństwie do tablicy imperatywnej, operacje modyfikujące składowe tablicy funkcyjnej nie nadpisują istniejącej tablicy, a tworzą jej kopię, przy czym oryginalna kopia nadal istnieje i może być używana w dalszych obliczeniach. Takie struktury sprawdzają się lepiej niż tablice imperatywne np. w algorytmach niedeterministycznych z nawrotami.

Rozważmy implementację tablic funkcyjnych za pomocą drzew binarnych postaci:

```
type 'a btree = Leaf | Node of 'a btree * 'a * 'a
btree
```

Zakładamy przy tym, że drzewo reprezentuje tablicę indeksowaną liczbami całkowitymi od 1 do n, a ścieżka do składowej o indeksie k, wyznaczona jest przez serię dzieleń modulo 2, aż do osiagnięcia wartości 1, wg zasady: jeśli k mod 2 = 0, to wybieramy lewego syna, a w przeciwnym razie - prawego, a następnie poszukujemy elementu o indeksie k div 2. W przypadku drzew zbalansowanych, a z takimi mamy tu do czynienia, dostęp do k-tego elementu wymaga log k kroków.

Zdefiniuj typ danych `'a array` (wraz z drzewem warto przechowywać najwyższy indeks w tablicy) oraz następujące operacje na tablicach funkcyjnych:

- `aempty : 'a array`, tablica pusta;
- `asub : 'a array -> int -> 'a`, pobranie składowej o zadanym indeksie;
- `aupdate : 'a array -> int -> 'a -> 'a array`, modyfikacja składowej o zadanym indeksie;
- `ahiext : 'a array -> 'a -> 'a array`, rozszerzenie tablicy o jedną składową;
- `ahirem : 'a array -> 'a array`, usunięcie składowej o najwyższym indeksie.

Zadanie 4 (2p.)

Chcemy w Ocamlu zdefiniować funkcję `sprintf` znaną z języka C, tak by np.

```
sprintf "Ala ma %d kot%s." : int -> string -> string
```

pozwalało zdefiniować funkcję

```
fun n -> sprintf "Ala ma %d kot%s." n (if n = 1 then "a" else if 1 < n & n < 5 then "y" else "ów")
```

Na pierwszy rzut oka wydaje się, że rozwiązanie tego zadania wymaga typów zależnych, ponieważ typ funkcji `sprintf` zależy od jej pierwszego argumentu. Okazuje się jednak, że polimorfizm parametryczny wystarczy. Dla uproszczenia załóżmy, że format nie jest zadany przez wartość typu `string` (nie chcemy zajmować się parsowaniem), ale przez konkatenację następujących dyrektyw formatujących:

- `lit s` - stała napisowa `s`
- `eol` - koniec wiersza
- `inr` - liczba typu `int`
- `flt` - liczba typu `float`
- `str` - napis typu `string`

Zakładając, że operatorem konkatenacji dyrektyw jest `++`, powyższy przykład może być zapisany następująco:

```
sprintf (lit "Ala ma " ++ inr ++ " kot" ++ str ++ lit ".") : int -> string -> string
```

Zdefiniuj funkcje `lit`, `eol`, `inr`, `flt`, `str`, `++` oraz funkcję `sprintf`. Podpowiedź: dyrektywy powinny być funkcjami transformującymi kontynuacje, a operator `++` to zwyczajne złożenie takich funkcji. Na przykład `inr` powinien mieć typ `(string -> a) -> string -> (int -> a)` (argumentem ma być kontynuacja oczekująca napisu, ale o nieokreślonym typie odpowiedzi, a wynikiem ma być kontynuacja oczekująca napisu, a następnie liczby całkowitej). Podobnie, typem `eol` ma być `(string -> a) -> string -> a`.

Zadanie 5 (6p.)

Przeanalizuj interpreter Prologa zamieszczony tutaj.

1. Zmień definicję funkcji `run` (i tylko tej funkcji) tak by interpreter liczył na ile spsobów dany cel może być spełniony przy danym programie zamiast sprawdzać tylko czy może być spełniony. (W interpreterze pojawi się nieogonowe wywołanie -- czy potrafisz zmodyfikować cały interpreter, tak by je wyeliminować?)
2. Rozważmy typ danych do reprezentowania wyrażeń regularnych:

```
type regexp =
 | Atom of char
 | And of regexp * regexp  (*  r1r2     *)
 | Or of regexp * regexp   (*  r1 | r2  *)
 | Star of regexp          (*  r*  *)
```

   Bazując na modelu obliczeń z nawrotami przy użyciu kontynuacji sukcesu oraz kontynuacji porażki, zaprezentowanym na przykładzie iterpretera Prologa, napisz funkcje

```
        match_regexp : regexp -> char list -> (char list -> (unit ->
'a) -> 'a) -> (unit -> 'a) -> 'a
```

   oraz

```
                        run : regexp -> char list -> bool
```

   które dla danego wyrażenia regularnego implementują niedeterministyczny automat rozpoznający język opisany przez to wyrażenie.