Kilka słów o wyjątkach

Wyjątki służą do sygnalizowania sytuacji wyjątkowych. Mechanizm ten jest bardzo podobny do występującego w C++. Użytkownik może deklarować własne rodzaje wyjątków i manipulować nimi jak innymi wartościami. Ponadto istnieje pewna liczba wyjątków standardowych, definiowanych przez język lub biblioteki.

```
# exception Pusta_lista;;
exception Pusta_lista

# let glowa l =
    match l with
      [] -> raise Pusta_lista
    |
      h::_ -> h ;;
val glowa : 'a list -> 'a = <fun>

# glowa [1;2;3];;
- : int = 1

# glowa [] ;;
Uncaught exception: Pusta_lista
```

Wyjątki są wykorzystywane przez funkcje biblioteczne do zgłaszania sytuacji błędnych.

```
# List.assoc 1 [(0, "zero"); (1, "jeden")];;
- : string = "jeden"

# List.assoc 3 [(0, "zero"); (1, "jeden")];;
Uncaught exception: Not_found
```

Wyjątki można obsługiwać za pomocą konstrukcji `try ... with`:

```
# let nazwa_cyfry_dwojkowej n =
    try
      List.assoc n [(0, "zero"); (1, "jeden")]
    with Not_found ->
      "to nie jest cyfra dwójkowa" ;;
val nazwa_cyfry_dwojkowej : int -> string = <fun>

# nazwa_cyfry_dwojkowej 0;;
- : string = "zero"

# nazwa_cyfry_dwojkowej 7;;
- : string = "to nie jest cyfra dwójkowa"
```

Konstrukcja `try ... with` pozwala na standardowe dopasowanie wzorców (tak jak `match ... with`).

```
# try
    1 / 0
  with
    x -> 1 ;;

# exception Blad of string;;
exception Blad of string

# let dziel_pare (a1, a2) (b1, b2) =
    if b1 = 0 then
      raise (Blad "b1 == 0")
    else if b2 = 0 then
      raise (Blad "b2 == 0")
    else
      (a1 / b1, a2 / b2) ;;
val dziel_pare : int * int ->
                 int * int -> int * int = <fun>

# dziel_pare (1, 0) (0, 1);;
Uncaught exception: Blad("a2 == 0")
```

Różniczkowanie symboliczne

Dla rozrywki napiszemy program do różniczkowania symbolicznego wielomianów.

```
# type expression =
    Const of float
  |
    Var of string
  |
    Sum of expression * expression
  |
    Diff of expression * expression
  |
    Prod of expression * expression
  |
    Quot of expression * expression ;;
```

Najpierw napiszemy funkcję, która, dostając wyrażenie i wartościowanie zmiennych, daje w wyniku wartość tego wyrażenia.

```
# exception Unbound_variable of string;;

# let rec eval env expr =
    match expr with
      Const f -> f
    |
      Var v ->
       (try
         List.assoc v env
       with
         Not_found -> raise (Unbound_variable v))
    |
      Sum (f, g) -> eval env f +. eval env g
    |
      Diff (f, g) -> eval env f -. eval env g
    |
      Prod (f, g) -> eval env f *. eval env g
    |
      Quot (f, g) -> eval env f /. eval env g ;;
val eval : (string * float) list
               -> expression -> float = <fun>
```

A teraz samo różniczkowanie:

```
# let rec deriv expr dv =
    match expr with
      Const _ -> Const 0.0
    |
      Var v ->
        if v = dv then Const 1.0 else Const 0.0
    |
      Sum (f, g) -> Sum (deriv f dv, deriv g dv)
    |
      Diff (f, g) -> Diff (deriv f dv, deriv g dv)
    |
      Prod (f, g) -> Sum (Prod(f, deriv g dv),
                          Prod(deriv f dv, g))
    |
      Quot (f, g) -> Quot (Diff(Prod(deriv f dv, g),
                                Prod(f, deriv g dv)),
                           Prod(g,g));;
val deriv : expression ->
              string -> expression = <fun>
```

Kilka zadań, żeby się nie nudzić

- Napisz funkcję `pelne a`, która buduje pełne drzewo binarne etykietowane wartością `a`. Jaki jest koszt działania tej funkcji?

- Drzewo jest *zrównoważone*, gdy każdy wierzchołek `Node (e, t1, t2)` spełnia warunek:

  $$\|\text{liczba\_wierzch } t1 - \text{liczba\_wierzch } t2\| \leq 1$$

  Napisz dwie funkcje: naiwną, która liczy głebokości każdego poddrzewa (duplikując pracę), i drugą, która unika powtarzania obliczeń.

- Napisz funckję `odbicie t1 t2`, która sprawdza, czy drzewo `t1` jest lustrzanym odbiciem drzewa `t2`. Uwaga: funkcja nie powinna budować nowych drzew, zatem nie powinna wywoływać konstruktora `Node` do budowy nowych wartości (ale może używać jej w dopasowaniu wzorców).

- Napisz zestaw funkcji, które będą wykonywać operacje arytmetyczne (dodawanie, odejmowanie, mnożenie, porównanie) na listach liczb, traktowanych jako liczby w zapisie dziesiętnym oraz funkcje konwersji między liczbami w tej notacji a liczbami typu `int`.
- Zdefiniuj typ danych dla drzew o zmiennej liczbie synów (i przykładowe operacje na nich). Podpowiedź: niech każdy wierzchołek ma *listę* synów (zamiast ustalonej ich liczby).