# Funktory aplikatywne
# &
# Parsery kombinarotoryczne

Przedstawiam dwa zadania do swojej prezentacji. Należy wykonać oba. Rozwiązania można przesyłać do końca listopada. Mój adres email: grzegorz314@gmail.com

## Zadanie teoretyczne.

W przypadku zwykłych list operator `(<*>)` ma typ `[a->b] -> [a] -> [b]`, a jego działanie polega na zaaplikowaniu każdej funkcji z lewego argumentu do każdej wartości z prawego argumentu. Moglibyśmy jednak wymagać innego zachowania: pierwsza funkcja z lewej listy jest aplikowana do pierwszej wartości z prawej listy, druga funkcja z lewej listy jest aplikowana do drugiej wartości z prawej listy, itd. Wówczas definicja byłaby następująca:

```
instance Applicative [] where
    pure x = ?????
    fs <*> xs = zipWith id fs xs
```

Sprawdź, że `pure x = [x]` nie spełnia praw funktorów aplikatywnych. Jak należy zdefiniować `pure`?

Wiadomo, że każda monada może zostać uczyniona instancją `Applicative` poprzez

```
pure = return
(<*>) = ap
```

Uzasadnij, że przedstawiona definicja listy jako funktora aplikatywnego nie jest monadyczna, tzn. nie da się uczynić listy instancją klasy `Monad` w taki sposób, żeby zachodziły powyższe równania.

## Zadanie praktyczne.

Rozważmy gramatykę, którą opisane są proste wyrażenia arytmetyczne:

```
expr ::= expr addop term | term
term ::= term multop factor | factor
factor ::= baseOrExp expop factor | baseOrExp
baseOrExp ::= integer | ( expr )
addop ::= + | -
multop ::= * | /
expop ::= **
```

Hierarchia priorytetów operatorów jest standardowa. Potęgowanie łączy w prawo, pozostałe operatory w lewo. W załączonym pliku znajduje się parser wyrażeń napisany w stylu monadycznym. Oto przykłady jego wywołania:

```
ghci> parse expr "8 / 2 / 2"
[(2,""),(4," / 2"),(8," / 2 / 2")]
ghci> parse expr "2  ** 3**2"
[(512,""),(8,"**2"),(2,"  ** 3**2")]
ghci> parse expr "2**3 - (5 - 3**2)"
[(12,""),(8," - (5 - 3**2)"),(2,"**3 - (5 - 3**2)")]
```

Zadanie polega na tym, żeby przepisać parser na styl aplikatywny. Trzeba usunąć kod określający `Parser` jako instancję klas `Monad` i `MonadPlus`, a zamiast tego uczynić `Parser` instancją `Applicative` i `Alternative`. W efekcie zamiast notacji `do` trzeba używać operatorów `(<$>)` i `(<*>)`. Oto przykład jak zmienić definicję parsera `word`:

```
word :: Parser String
word = aux <+> result "" where
  aux = do
      x <- letter
      xs <- word
      return (x:xs)
```

```
word :: Parser String
word = aux <|> result "" where
  aux = (:) <$> letter <*> word
```

**Uwaga.** Parser `sat :: (Char -> Bool) -> Parser Char` parsuje kolejny znak tylko jeśli spełnia on podany predykat. Osobiście nie potrafię napisać go korzystając tylko z atomowych parserów `result`, `zero`, `item` oraz operatorów `(<$>)` i `(<*>)`. Jeżeli komuś się to uda, to dostanie dodatkowe punkty :). Proponuję jednak, żeby dopisać sobie parser `satisfy :: Parser a -> (a -> Bool) -> Parser a`, który dla podanego parsera `p` i predykatu $\varphi$ akceptuje tylko te wyniki parsera `p`, które spełniają predykat $\varphi$. Ten parser także uznajemy za atomowy. Wszystkie pozostałe parsery powinny być zapisane bez wnikania w strukturę typu `Parser`, zamiast tego należy używać wcześniej zdefiniowanych parserów i operatorów `(<$>)` i `(<*>)`. W szczególności oznacza to, że linia "`Parser $ \inp ->`" jest niemile widziana poza definicjami parserów atomowych oraz instancjami klas `Functor`, `Applicative`, `Alternative`.