Podstawy Programowania Funkcyjnego

Chlebowski Sz. Gajda A.

Zakład Logiki i Kognitywistyki
Instytut Psychologii
Uniwersytet im. Adama Mickiewicza

25.02.2020

Plan na dziś

1. Informacje organizacyjne
2. Paradygmaty programowania
3. Historia programowania funkcyjnego
4. Haskell — wprowadzenie

Informacje organizacyjne

- 60 godzin laboratorium, 8 ECTS

Forma aktywności	Średnia liczba godzin
Godziny zajęciowe	60
Praca własna studenta: przygotownaie do zajęć	40
Praca własna studenta: zapoznanie się z literaturą przedmiotu	10
Praca własna studenta: przygotowanie projektów	90

Przedmiot dzieli się na 3 moduły

1. Podstawy języka Haskell
   Projekt 1
2. Rozszerzone zagadnienia dotyczące Haskell'a
   Projekt 2
3. Podstawy języka Scala
   Projekt 3

Efekty kształcenia

Student(ka)...

1. Zna i rozumie różnice pomiędzy paradygmatami programowania.

Efekty kształcenia

Student(ka)...

1. Zna i rozumie różnice pomiędzy paradygmatami programowania.
2. Zna genezę powstania języków funkcyjnych.

Efekty kształcenia

Student(ka)...

1. Zna i rozumie różnice pomiędzy paradygmatami programowania.
2. Zna genezę powstania języków funkcyjnych.
3. Rozumie pojęcie rekurencji, potrafi rekurencyjnie zdefiniować funkcję.

Efekty kształcenia

Student(ka)...

1. Zna i rozumie różnice pomiędzy paradygmatami programowania.
2. Zna genezę powstania języków funkcyjnych.
3. Rozumie pojęcie rekurencji, potrafi rekurencyjnie zdefiniować funkcję.
4. Rozumie różnice pomiędzy różnymi strategiami obliczania funkcji, zna zalety i wady każdej z nich.

Efekty kształcenia

Student(ka)...

1. Zna i rozumie różnice pomiędzy paradygmatami programowania.
2. Zna genezę powstania języków funkcyjnych.
3. Rozumie pojęcie rekurencji, potrafi rekurencyjnie zdefiniować funkcję.
4. Rozumie różnice pomiędzy różnymi strategiami obliczania funkcji, zna zalety i wady każdej z nich.
5. Wie czym jest program napisany w Haskellu i rozumie jego strukturę.

Efekty kształcenia

Student(ka)...

1. Zna i rozumie różnice pomiędzy paradygmatami programowania.
2. Zna genezę powstania języków funkcyjnych.
3. Rozumie pojęcie rekurencji, potrafi rekurencyjnie zdefiniować funkcję.
4. Rozumie różnice pomiędzy różnymi strategiami obliczania funkcji, zna zalety i wady każdej z nich.
5. Wie czym jest program napisany w Haskellu i rozumie jego strukturę.
6. Rozumie pojęcie typu, klasy oraz metody, potrafi samodzielnie określić typ określonych wyrażeń oraz sprawdzić, czy jego/jej odpowiedź jest poprawna.

Efekty kształcenia

Student(ka)...

1. Zna i rozumie różnice pomiędzy paradygmatami programowania.
2. Zna genezę powstania języków funkcyjnych.
3. Rozumie pojęcie rekurencji, potrafi rekurencyjnie zdefiniować funkcję.
4. Rozumie różnice pomiędzy różnymi strategiami obliczania funkcji, zna zalety i wady każdej z nich.
5. Wie czym jest program napisany w Haskellu i rozumie jego strukturę.
6. Rozumie pojęcie typu, klasy oraz metody, potrafi samodzielnie określić typ określonych wyrażeń oraz sprawdzić, czy jego/jej odpowiedź jest poprawna.
7. Potrafi zaprojektować aplikację w języku Haskell spełniającą określone wstępnie kryteria.

Efekty kształcenia

Student(ka)...

1. Zna i rozumie różnice pomiędzy paradygmatami programowania.
2. Zna genezę powstania języków funkcyjnych.
3. Rozumie pojęcie rekurencji, potrafi rekurencyjnie zdefiniować funkcję.
4. Rozumie różnice pomiędzy różnymi strategiami obliczania funkcji, zna zalety i wady każdej z nich.
5. Wie czym jest program napisany w Haskellu i rozumie jego strukturę.
6. Rozumie pojęcie typu, klasy oraz metody, potrafi samodzielnie określić typ określonych wyrażeń oraz sprawdzić, czy jego/jej odpowiedź jest poprawna.
7. Potrafi zaprojektować aplikację w języku Haskell spełniającą określone wstępnie kryteria.
8. Potrafi poruszać się w środowiskach służących do zarządzania projektami programistycznymi w języku Haskell.

Efekty kształcenia

Student(ka)...

1. Zna i rozumie różnice pomiędzy paradygmatami programowania.
2. Zna genezę powstania języków funkcyjnych.
3. Rozumie pojęcie rekurencji, potrafi rekurencyjnie zdefiniować funkcję.
4. Rozumie różnice pomiędzy różnymi strategiami obliczania funkcji, zna zalety i wady każdej z nich.
5. Wie czym jest program napisany w Haskellu i rozumie jego strukturę.
6. Rozumie pojęcie typu, klasy oraz metody, potrafi samodzielnie określić typ określonych wyrażeń oraz sprawdzić, czy jego/jej odpowiedź jest poprawna.
7. Potrafi zaprojektować aplikację w języku Haskell spełniającą określone wstępnie kryteria.
8. Potrafi poruszać się w środowiskach służących do zarządzania projektami programistycznymi w języku Haskell.
9. Wie czym jest program napisany w Scali i rozumie jego strukturę.

Punktacja

- 3 projekty — każdy po 20 pkt.
- 6 wej/wyjściówek — kazda po 3 pkt.
- łącznie można uzyskać 78 pkt.

Suma punktów	**%**	**Ocena**
0–<46	0–<60	**2**
46–<52	60–<67	**3**
52–<57	67–<74	**3,5**
57–<63	74–<82	**4**
63–<70	82–<90	**4,5**
70	90	**5**

Punktacja

W przypadku nieuzyskania wystarczającej liczby punktów, możliwe jest napisanie projektu poprawkowego, który obejmuje materiał ze wszystkich modułów (punkty uzyskane w ramach wejściówek/wyjściówek są przepisywane). Indywidualnych projektów poprawiać nie można.

Kontakt

- Szymon Chlebowski
  - e-mail: szymon.chlebowski@amu.edu.pl
  - dyżur: czwartek 13:00–14:00, pok. 97 w budynku AB
- Andrzej Gajda
  - strona internetowa: http://ag96820.home.amu.edu.pl
  - e-mail: andrzej.gajda@amu.edu.pl
  - dyżur: czwartek 9:00–10:00, lab. RRG w budynku AB

Literatura

- Graham Hutton. *Programming in Haskell*, Cambridge University Press, Cambridge 2007.
- Martin Odersky, Lex Spoon, Bill Venners. *Programming in Scala*, Artima 2008.
  (edycja pierwsza dostępna na: `http://www.artima.com/pins1ed/`)
- Mark C. Lewis, Lisa L. Lacher, *Introduction to Programming and Problem-Solving Using Scala*, CRC Press, 2017.
- Benjamin Pierce. *Types and Programming Languages*, The MIT Press, Cambridge 2002.

Paradygmaty programowania

- programowanie imperatywne (języki: *C*, *Python*, *Java*)
- programowanie logiczne (języki: *Prolog*, *Datalog*)
- programowanie obiektowe (języki: *Java*, *Python*, *Scala*)
- programowanie funkcyjne (języki: *Haskell*, *Scala*)

⋮

- Program postrzegany jest jako ciąg poleceń dla komputera.
- Wykonanie programu rozumiemy tu jako sekwencję poleceń zmieniających krok po kroku stan maszyny, aż do uzyskania oczekiwanego wyniku.
- Podstawą są tutaj tradycyjne petle (*while*, *for*, itd.) oraz użycie zmiennych do ich kontroli.

- Wykonanie programu to próba udowodnienia celu w oparciu o podane informacje i reguły.
- Nie „wydajemy rozkazów” (jak w paradygmacie imperetywnym), a jedynie opisujemy, co wiemy i co chcemy uzyskać.

```
ojciec(jan, jerzy).
ojciec(jerzy, janusz).
ojciec(jerzy, piotr).
dziadek(X, Z) :- ojciec(X, Y), ojciec(Y, Z).

?- dziadek(X, janusz).
```

- Połączenie danych/stanu z operacjami na nich (metodami/poleceniami) w całość, stanowiącą odrębną jednostkę — obiekt.
- Obiekty komunikują się ze sobą.
- Obecny jest mechanizm dziedziczenia — możliwośc tworzenia wyspecjalizowanych obiektów w oparciu o obiekty ogólne.

- Brak zmiennych.
- Skoro nie ma zmiennych, nie można kontrolować zachowań pętli.
- Brak tradycyjnie rozumianych petli.
- Program jest złożeniem funkcji, gdzie termin ‘złożenie’ i ‘funkcja’ rozumiane są tak, jak w matematyce.
- Obliczenia polegają na zastosowaniu funkcji do argumentu (odpowiada to podstawowej relacji w rachunku lambda — β–redukcji).
- Rekurencja i funkcje wyższego rzędu są wszechobecne.

- Podstawowa intuicja jest taka: wartość funkcji dla pewnej klasy argumentów jest określana przy użyciu wartości tej funkcji dla pewnych argumentów.

```
pred(0)   = 0
pred(1)   = 0
pred(n)   = pred(n-1) + 1, dla n >= 2
```

```
fib(0)   = 1
fib(1)   = 1
fib(n)   = fib(n-2) + fib(n-1), dla n >= 2
```

Historia programowania funkcyjnego

- Alonzo Church (1903–1995)
- Rachunek lambda, nierozstrzygalność logiki pierwszego rzędu
- Twórca szkoły teorii rekursji (Kleene, Rosser, Curry)
- **Haskell** Curry
- **Church's Thesis**: *Effectively computable functions from positive integers to positive integers are just those definable in the lambda calculus.*
- Rachunek lambda z typami vs rachunek lambda bez typów

- W teorii mnogości funkcje są specyficznymi relacjami, są zatem zbiorami par uporządkowanych.
- Czy funkcje $f(x) = x + 2 - 1$ oraz $g(x) = x + 1$ są takie same?
- W rachunku lambda funkcje nie są zbiorami par uporządkowanych, są pewnymi obiektami, mającymi określony *typ*.

- Główne idee:

- Główne idee:
  - zastosowanie funkcji/aplikacja do argumentu

- Główne idee:
  - zastosowanie funkcji/aplikacja do argumentu
  - tworzenie funkcji przez abstrakcje

- Główne idee:
  - zastosowanie funkcji/aplikacja do argumentu
  - tworzenie funkcji przez abstrakcje
- Funkcje jako reguły obliczania, przykład:

- Główne idee:
  - zastosowanie funkcji/aplikacja do argumentu
  - tworzenie funkcji przez abstrakcje
- Funkcje jako reguły obliczania, przykład:
  - funkcja: $x^2 - 2x + 5$

- Główne idee:
  - zastosowanie funkcji/aplikacja do argumentu
  - tworzenie funkcji przez abstrakcje
- Funkcje jako reguły obliczania, przykład:
  - funkcja: $x^2 - 2x + 5$
  - jako obiekt: $\lambda x.x^2 - 2x + 5$

- Główne idee:
  - zastosowanie funkcji/aplikacja do argumentu
  - tworzenie funkcji przez abstrakcje
- Funkcje jako reguły obliczania, przykład:
  - funkcja: $x^2 - 2x + 5$
  - jako obiekt: $\lambda x.x^2 - 2x + 5$

Jeśli 'zastosujemy' obiekt '$\lambda x.x^2 - 2x + 5$' do liczby 2 otrzymamy

$$2^2 - 2 * 2 + 5 = 4 - 4 + 5 = 5$$

Kiedy pojawia się argument, lambda znika.

- Główne idee:
  - zastosowanie funkcji/aplikacja do argumentu
  - tworzenie funkcji przez abstrakcje
- Funkcje jako reguły obliczania, przykład:
  - funkcja: $x^2 - 2x + 5$
  - jako obiekt: $\lambda x.x^2 - 2x + 5$

Jeśli 'zastosujemy' obiekt '$\lambda x.x^2 - 2x + 5$' do liczby 2 otrzymamy

$$2^2 - 2 * 2 + 5 = 4 - 4 + 5 = 5$$

Kiedy pojawia się argument, lambda znika.

Typ funkcji $\lambda x.x^2 - 2x + 5$: argumentem i wartością funkcji są liczby. Zasosowanie tej funkcji do argumentów innego typu nie ma sensu.

Języki funkcyjne — przykłady (1/2)

- Rachunek lambda stanowi podstawę wszystkich języków funkcyjnych.
- Jednym z pierwszych takich języków był *LISP* (*LISt Processor*, John McCarthy — późne lata 50-te).
  - Po raz pierwszy użyto lambda-termów, funkcję '$\lambda x.e$' zapisywanych jako '*lambda (x) e*'.
  - Wprowadzenie list i funkcji wyższego rzędu, które na nich operują.
  - Wprowadzenie wyrażeń warunkowych używanych w definiowaniu funkcji przez rekurencje.

```
(define mapcar (fun lst)
 (if (null lst)
   nil
   (cons (fun (car lst)) (mapcar fun (cdr lst))) ))
```

- Kolejny wpływowym językiem funckyjnym był *FP* (*Functional Programming*, John Backus, lata 70-te)
- Kolejny język to *ML* (*Meta Language*, Robin Milner, lata 70-te).
  - *ML* jest czasem charakteryzowany jako ‘Lisp z typami’.
  - Wprowadzenie *type inference* oraz strategii ewaluacji programu *call-by-value*.

```
fun fac (0 : int) : int = 1
  | fac (n : int) : int = n * fac (n - 1)
```

- *Haskell* jest czysto funkcyjnym językiem programowania (nazwa pochodzi od nazwiska logika: Haskell Curry, uczeń Alonzo Church'a).
- Jest efektem pracy wielu badaczy (m.in. Paul Hudak, Simon Peyton Jones, Philip Wadler).

- *Zwarte programy* (*concise programs*): z uwagi na powszechne używanie funkcji wyższego rzędu, programy napisane w języku Haskell są zwykle krótsze niż ich imperatywne odpowiedniki (np. *qsort*).

Haskell — cechy (1/4)

- *Zwarte programy* (*concise programs*): z uwagi na powszechne używanie funkcji wyższego rzędu, programy napisane w języku Haskell są zwykle krótsze niż ich imperatywne odpowiedniki (np. *qsort*).
- *Zaawansowany system typów* (*powerful type system*): z uwagi na wyrafinowany system typów, bardzo wiele błędów jest rozpoznawanych jeszcze przed uruchomieniem programu (przed zastosowaniem funkcji). System typów języka Haskell wspiera *polimorfizm*.

- *Zwarte programy* (*concise programs*): z uwagi na powszechne używanie funkcji wyższego rzędu, programy napisane w języku Haskell są zwykle krótsze niż ich imperatywne odpowiedniki (np. *qsort*).
- *Zaawansowany system typów* (*powerful type system*): z uwagi na wyrafinowany system typów, bardzo wiele błędów jest rozpoznawanych jeszcze przed uruchomieniem programu (przed zastosowaniem funkcji). System typów języka Haskell wspiera *polimorfizm*.
- *Definiowanie przez abstrakcje* (*list comprehension*): język Haskell posiada specjalną notację do definiowania list elementów, poprzez filtrowanie innych list. W ten sposób można w bardzo zwięzły sposób definiować skomplikowane kolekcje. Przykład:
  $f = \{(i,j) \mid i \in \{1,2\},\ j \in \{1,2,3,4\}\}$

```
f = [(i,j) | i <- [1,2], j <- [1..4]]

[(1,1),(1,2),(1,3),(1,4),(2,1),(2,2),(2,3),(2,4)]
```

- *Funkcje rekurencyjne* (*Recursive functions*): większość nietrywialnych programów wymaga pewnego rodzaju pętli. W języku Haskell jednym z podstawowych mechanizmów konstruowania pętli jest rekurencja, polegająca (ogólnie mówiąc) na definiowaniu funkcji przy użyciu ich samych.

- *Funkcje rekurencyjne* (*Recursive functions*): większość nietrywialnych programów wymaga pewnego rodzaju pętli. W języku Haskell jednym z podstawowych mechanizmów konstruowania pętli jest rekurencja, polegająca (ogólnie mówiąc) na definiowaniu funkcji przy użyciu ich samych.
- *Funkcje wyższego rzędu* (*Higher-order functions*): w języku Haskell funkcje mogą przyjmować jako argumenty (i zwracać jako wartości) inne dowolne funkcje. Dzięki temu abstrakcyjne operacje, takie jak złożenie dwóch funkcji, mogą być definiowane na bardzo ogólnym poziomie i mogą być używane w bardzo różnych programach.

```
compose :: (b -> c) -> (a -> b) -> a -> c
compose f g  = \x -> f (g x)
```

- *Efekty uboczne* (*Effectful functions*): w języku Haskell funkcje są czyste, i.e., przyjmują one pewne argumenty i zwracają wartości, natomiast nie ma do nich dostępu, kiedy już zostaną wywołane. Tego rodzaju zachowanie bardzo ułatwia dowody tego, że program działa poprawnie. Haskell posiada system pozwalający na zarządzanie efektami ubocznymi, takimi jak wyświetlanie pośrednich rezultatów działania funkcji na ekranie, bez poświęcania zasady, że wszystkie funkcje muszą być czyste.

- *Efekty uboczne* (*Effectful functions*): w języku Haskell funkcje są czyste, i.e., przyjmują one pewne argumenty i zwracają wartości, natomiast nie ma do nich dostępu, kiedy już zostaną wywołane. Tego rodzaju zachowanie bardzo ułatwia dowody tego, że program działa poprawnie. Haskell posiada system pozwalający na zarządzanie efektami ubocznymi, takimi jak wyświetlanie pośrednich rezultatów działania funkcji na ekranie, bez poświęcania zasady, że wszystkie funkcje muszą być czyste.
- Funkcje uogólnione (*Generic functions*): system typów języka Haskell pozwala na definiowanie funkcji, która akceptują argumenty bardzo róznych typów.

- *Leniwa ewaluacja, call-by-name* (*Lazy evaluation*): w jezyku Haskell programy są wykonywane ‘leniwie’, i.e., funkcje są obliczane, o ile ich rezultaty są rzeczywiście niezbędne. Pozwala to na definiowanie potencjalnie nieskończonych struktur, których tyko skończone fragmenty są używane.

```
take 5 [1..] = [1,2,3,4,5]
```

- *Leniwa ewaluacja, call-by-name* (*Lazy evaluation*): w jezyku Haskell programy są wykonywane ‘leniwie’, i.e., funkcje są obliczane, o ile ich rezultaty są rzeczywiście niezbędne. Pozwala to na definiowanie potencjalnie nieskończonych struktur, których tyko skończone fragmenty są używane.

```
take 5 [1..] = [1,2,3,4,5]
```

- *Dowodzenie poprawności programów* (*equational reasoning*): dzięki temu, że wszystkie funkcje są czyste, a każdy program jest złożeniem funkcji, dowodzenie poprawności programów jest względnie proste i sprowadza się do ciągu równości. Pożyteczną techniką jest również rozumowanie przez indukcję.

- *Leniwa ewaluacja, call-by-name* (*Lazy evaluation*): w jezyku Haskell programy są wykonywane ‘leniwie’, i.e., funkcje są obliczane, o ile ich rezultaty są rzeczywiście niezbędne. Pozwala to na definiowanie potencjalnie nieskończonych struktur, których tyko skończone fragmenty są używane.

```
take 5 [1..] = [1,2,3,4,5]
```

- *Dowodzenie poprawności programów* (*equational reasoning*): dzięki temu, że wszystkie funkcje są czyste, a każdy program jest złożeniem funkcji, dowodzenie poprawności programów jest względnie proste i sprowadza się do ciągu równości. Pożyteczną techniką jest również rozumowanie przez indukcję.
- *Definiowanie nowych typów*: język Haskell wspiera technikę definiowania nowych typów przez użytkownika. Można w ten sposób tworzyć własny język typów i okreslać funkcję, które z nich korzystają.

Haskell — wprowadzenie

Funkcje — przykład 1

```
double x = x + x
```

Funkcje — przykład 1

```
double x = x + x
```

double 3

Funkcje — przykład 1

```
double x = x + x
```

double 3

= { *stosowanie* funkcji ‘double’ do argumentu ‘3’ }

Funkcje — przykład 1

```
double x = x + x
```

double 3

$=$ { *stosowanie* funkcji ‘double’ do argumentu ‘3’ }

$3 + 3$

Funkcje — przykład 1

```
double x = x + x
```

$$
\begin{aligned}
& \texttt{double}\ 3 \\
= & \ \{\ \textit{stosowanie}\ \text{funkcji `double' do argumentu `3'}\ \} \\
& 3 + 3 \\
= & \ \{\ \textit{stosowanie}\ \text{funkcji `}+\text{' do argumentów `3' i `3'}\ \}
\end{aligned}
$$

Funkcje — przykład 1

```
double x = x + x
```

double 3

= { *stosowanie* funkcji ‘double’ do argumentu ‘3’ }

3 + 3

= { *stosowanie* funkcji ‘+’ do argumentów ‘3’ i ‘3’ }

6

Funkcje — przykład 2

```
double x = x + x
```

double (double 2)

Funkcje — przykład 2

```
double x = x + x
```

double (double 2)

$=$ { *stosowanie* funkcji ‘double’ do argumentu ‘2’ }

Funkcje — przykład 2

```
double x = x + x
```

$$
\begin{aligned}
& \text{double (double 2)} \\
= \{\ & \textit{stosowanie} \text{ funkcji 'double' do argumentu '2'} \ \} \\
& \text{double } (2 + 2)
\end{aligned}
$$

Funkcje — przykład 2

```
double x = x + x
```

$$
\begin{aligned}
& \text{double (double 2)} \\
= & \{ \textit{stosowanie} \text{ funkcji ‘double’ do argumentu ‘2’ } \} \\
& \text{double (2 + 2)} \\
= & \{ \textit{stosowanie} \text{ funkcji ‘+’ do argumentów ‘2’ i ‘2’ } \}
\end{aligned}
$$

Funkcje — przykład 2

```
double x = x + x
```

double (double 2)

= { *stosowanie* funkcji ‘double’ do argumentu ‘2’ }

double (2 + 2)

= { *stosowanie* funkcji ‘+’ do argumentów ‘2’ i ‘2’ }

double 4

Funkcje — przykład 2

```
double x = x + x
```

double (double 2)

= { *stosowanie* funkcji ‘double’ do argumentu ‘2’ }

double (2 + 2)

= { *stosowanie* funkcji ‘+’ do argumentów ‘2’ i ‘2’ }

double 4

= { *stosowanie* funkcji ‘double’ do argumentu ‘4’ }

Funkcje — przykład 2

```
double x = x + x
```

double (double 2)

= { *stosowanie* funkcji ‘double’ do argumentu ‘2’ }

double (2 + 2)

= { *stosowanie* funkcji ‘+’ do argumentów ‘2’ i ‘2’ }

double 4

= { *stosowanie* funkcji ‘double’ do argumentu ‘4’ }

4 + 4

```
double x = x + x
```

double (double 2)

= { *stosowanie* funkcji ‘double’ do argumentu ‘2’ }

double (2 + 2)

= { *stosowanie* funkcji ‘+’ do argumentów ‘2’ i ‘2’ }

double 4

= { *stosowanie* funkcji ‘double’ do argumentu ‘4’ }

4 + 4

= { *stosowanie* funkcji ‘+’ do argumentów ‘4’ i ‘4’ }

```
double x = x + x
```

double (double 2)

= { *stosowanie* funkcji ‘double’ do argumentu ‘2’ }

double (2 + 2)

= { *stosowanie* funkcji ‘+’ do argumentów ‘2’ i ‘2’ }

double 4

= { *stosowanie* funkcji ‘double’ do argumentu ‘4’ }

4 + 4

= { *stosowanie* funkcji ‘+’ do argumentów ‘4’ i ‘4’ }

8

Funkcje — przykład 2'

```
double x = x + x
```

double (double 2)

Funkcje — przykład 2'

```
double x = x + x
```

double (double 2)

= { *stosowanie* funkcji 'double' do argumentu 'double 2' }

Funkcje — przykład 2'

```
double x = x + x
```

double (double 2)

= { *stosowanie* funkcji 'double' do argumentu 'double 2' }

double 2 + double 2

Funkcje — przykład 2'

```
double x = x + x
```

double (double 2)

= { *stosowanie* funkcji 'double' do argumentu 'double 2' }

double 2 + double 2

= { *stosowanie* funkcji 'double' do argumentu '2' }

Funkcje — przykład 2'

```
double x = x + x
```

double (double 2)

= { *stosowanie* funkcji 'double' do argumentu 'double 2' }

double 2 + double 2

= { *stosowanie* funkcji 'double' do argumentu '2' }

(2 + 2) + double 2

Funkcje — przykład 2'

```
double x = x + x
```

double (double 2)

= { *stosowanie* funkcji 'double' do argumentu 'double 2' }

double 2 + double 2

= { *stosowanie* funkcji 'double' do argumentu '2' }

(2 + 2) + double 2

= { *stosowanie* funkcji '+' do argumentów '2' i '2' }

Funkcje — przykład 2'

```
double x = x + x
```

double (double 2)

= { *stosowanie* funkcji 'double' do argumentu 'double 2' }

double 2 + double 2

= { *stosowanie* funkcji 'double' do argumentu '2' }

(2 + 2) + double 2

= { *stosowanie* funkcji '+' do argumentów '2' i '2' }

4 + double 2

```
double x = x + x
```

double (double 2)

= { *stosowanie* funkcji 'double' do argumentu 'double 2' }

double 2 + double 2

= { *stosowanie* funkcji 'double' do argumentu '2' }

(2 + 2) + double 2

= { *stosowanie* funkcji '+' do argumentów '2' i '2' }

4 + double 2

= { *stosowanie* funkcji 'double' do argumentu '2' }

```
double x = x + x
```

double (double 2)

= { *stosowanie* funkcji 'double' do argumentu 'double 2' }

double 2 + double 2

= { *stosowanie* funkcji 'double' do argumentu '2' }

(2 + 2) + double 2

= { *stosowanie* funkcji '+' do argumentów '2' i '2' }

4 + double 2

= { *stosowanie* funkcji 'double' do argumentu '2' }

4 + (2 + 2)

```
double x = x + x
```

double (double 2)

= { *stosowanie* funkcji 'double' do argumentu 'double 2' }

double 2 + double 2

= { *stosowanie* funkcji 'double' do argumentu '2' }

(2 + 2) + double 2

= { *stosowanie* funkcji '+' do argumentów '2' i '2' }

4 + double 2

= { *stosowanie* funkcji 'double' do argumentu '2' }

4 + (2 + 2)

= { *stosowanie* funkcji '+' do argumentów '2' i '2' }

```
double x = x + x
```

double (double 2)

= { *stosowanie* funkcji 'double' do argumentu 'double 2' }

double 2 + double 2

= { *stosowanie* funkcji 'double' do argumentu '2' }

(2 + 2) + double 2

= { *stosowanie* funkcji '+' do argumentów '2' i '2' }

4 + double 2

= { *stosowanie* funkcji 'double' do argumentu '2' }

4 + (2 + 2)

= { *stosowanie* funkcji '+' do argumentów '2' i '2' }

4 + 4

Funkcje — przykład 2'

```
double x = x + x
```

double (double 2)

= { *stosowanie* funkcji 'double' do argumentu 'double 2' }

double 2 + double 2

= { *stosowanie* funkcji 'double' do argumentu '2' }

(2 + 2) + double 2

= { *stosowanie* funkcji '+' do argumentów '2' i '2' }

4 + double 2

= { *stosowanie* funkcji 'double' do argumentu '2' }

4 + (2 + 2)

= { *stosowanie* funkcji '+' do argumentów '2' i '2' }

4 + 4

= { *stosowanie* funkcji '+' do argumentów '4' i '4' }

Funkcje — przykład 2'

```
double x = x + x
```

double (double 2)

= { *stosowanie* funkcji 'double' do argumentu 'double 2' }

double 2 + double 2

= { *stosowanie* funkcji 'double' do argumentu '2' }

(2 + 2) + double 2

= { *stosowanie* funkcji '+' do argumentów '2' i '2' }

4 + double 2

= { *stosowanie* funkcji 'double' do argumentu '2' }

4 + (2 + 2)

= { *stosowanie* funkcji '+' do argumentów '2' i '2' }

4 + 4

= { *stosowanie* funkcji '+' do argumentów '4' i '4' }

8

Funkcje — przykład 3 (imperatywny)

Suma liczb od 1 do n

```
int total = 0
for (int count = 1; count <= n; count++)
    total = total + count
```

Dla $n = 5$:

Funkcje — przykład 3 (imperatywny)

Suma liczb od 1 do n

```
int total = 0
for (int count = 1; count <= n; count++)
    total = total + count
```

Dla $n = 5$:

total $= 0$; count $= 1$

Funkcje — przykład 3 (imperatywny)

Suma liczb od 1 do n

```
int total = 0
for (int count = 1; count <= n; count++)
    total = total + count
```

Dla $n = 5$:

total $= 0$; count $= 1$
total $= 1$; count $= 2$

Funkcje — przykład 3 (imperatywny)

Suma liczb od 1 do n

```
int total = 0
for (int count = 1; count <= n; count++)
    total = total + count
```

Dla $n = 5$:

total $= 0$; count $= 1$
total $= 1$; count $= 2$
total $= 3$; count $= 3$

Suma liczb od 1 do n

```
int total = 0
for (int count = 1; count <= n; count++)
    total = total + count
```

Dla $n = 5$:

total $= 0$; count $= 1$
total $= 1$; count $= 2$
total $= 3$; count $= 3$
total $= 6$; count $= 4$

Suma liczb od 1 do n

```
int total = 0
for (int count = 1; count <= n; count++)
    total = total + count
```

Dla $n = 5$:

total $= 0$; count $= 1$
total $= 1$; count $= 2$
total $= 3$; count $= 3$
total $= 6$; count $= 4$
total $= 10$; count $= 5$

Funkcje — przykład 3 (imperatywny)

Suma liczb od 1 do n

```
int total = 0
for (int count = 1; count <= n; count++)
    total = total + count
```

Dla $n = 5$:

total $= 0$; count $= 1$
total $= 1$; count $= 2$
total $= 3$; count $= 3$
total $= 6$; count $= 4$
total $= 10$; count $= 5$
total $= 15$

Funkcje — przykład 3'

```
sum  [1..n]
```

Dla $n = 5$

Funkcje — przykład 3'

```
sum  [1..n]
```

Dla $n = 5$

sum [1..5]

Funkcje — przykład 3'

```
sum  [1..n]
```

Dla $n = 5$

$$
\begin{aligned}
& \text{sum } [1..5] \\
= & \{ \textit{stosowanie} \text{ funkcji '[..]'} \}
\end{aligned}
$$

```
sum  [1..n]
```

Dla $n = 5$

$$
\begin{aligned}
& \texttt{sum [1..5]} \\
= & \{ \textit{stosowanie} \text{ funkcji '[..]'} \} \\
& \texttt{sum [1, 2, 3, 4, 5]}
\end{aligned}
$$

```
sum  [1..n]
```

Dla $n = 5$

$$
\begin{aligned}
& \texttt{sum [1..5]} \\
= & \{ \textit{stosowanie} \text{ funkcji '[..]'} \} \\
& \texttt{sum [1, 2, 3, 4, 5]} \\
= & \{ \textit{stosowanie} \text{ funkcji 'sum'} \}
\end{aligned}
$$

```
sum  [1..n]
```

Dla $n = 5$

$$
\begin{aligned}
& \texttt{sum}\ [1..5] \\
= & \{\ \textit{stosowanie}\ \text{funkcji `[..]'}\ \} \\
& \texttt{sum}\ [1, 2, 3, 4, 5] \\
= & \{\ \textit{stosowanie}\ \text{funkcji `sum'}\ \} \\
& 1 + 2 + 3 + 4 + 5
\end{aligned}
$$

```
sum  [1..n]
```

Dla $n = 5$

$$
\begin{aligned}
& \texttt{sum [1..5]} \\
= & \{ \textit{stosowanie} \text{ funkcji '[..]'} \} \\
& \texttt{sum [1, 2, 3, 4, 5]} \\
= & \{ \textit{stosowanie} \text{ funkcji 'sum'} \} \\
& 1 + 2 + 3 + 4 + 5 \\
= & \{ \textit{stosowanie} \text{ funkcji '+' do argumentów '1', '2', '3', '4' i '5'} \}
\end{aligned}
$$

```
sum  [1..n]
```

Dla $n = 5$

sum [1..5]

= { *stosowanie* funkcji '[..]' }

sum [1, 2, 3, 4, 5]

= { *stosowanie* funkcji 'sum' }

$1 + 2 + 3 + 4 + 5$

= { *stosowanie* funkcji '+' do argumentów '1', '2', '3', '4' i '5' }

15

Funkcje — przykład 3”

```
sum []       = 0
sum (n:ns)   = n + sum ns
```

sum [1, 2, 3]

```
sum []     = 0
sum (n:ns) = n + sum ns
```

sum [1, 2, 3]

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[1, 2, 3]’ }

```
sum []       = 0
sum (n:ns)   = n + sum ns
```

sum [1, 2, 3]

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[1, 2, 3]’ }

1 + sum [2, 3]

Funkcje — przykład 3”

```
sum []       = 0
sum (n:ns)   = n + sum ns
```

sum [1, 2, 3]

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[1, 2, 3]’ }

1 + sum [2, 3]

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[2, 3]’ }

Funkcje — przykład 3”

```
sum []       = 0
sum (n:ns)   = n + sum ns
```

sum [1, 2, 3]

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[1, 2, 3]’ }

1 + sum [2, 3]

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[2, 3]’ }

1 + (2 + sum [3])

Funkcje — przykład 3”

```
sum []     = 0
sum (n:ns) = n + sum ns
```

sum [1, 2, 3]

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[1, 2, 3]’ }

1 + sum [2, 3]

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[2, 3]’ }

1 + (2 + sum [3])

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[3]’ }

Funkcje — przykład 3”

```
sum []     = 0
sum (n:ns) = n + sum ns
```

sum [1, 2, 3]

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[1, 2, 3]’ }

1 + sum [2, 3]

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[2, 3]’ }

1 + (2 + sum [3])

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[3]’ }

1 + (2 + (3 + sum []))

```
sum []       = 0
sum (n:ns)   = n + sum ns
```

sum [1, 2, 3]

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[1, 2, 3]’ }

1 + sum [2, 3]

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[2, 3]’ }

1 + (2 + sum [3])

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[3]’ }

1 + (2 + (3 + sum []))

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[]’ }

```
sum []       = 0
sum (n:ns)   = n + sum ns
```

sum [1, 2, 3]

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[1, 2, 3]’ }

1 + sum [2, 3]

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[2, 3]’ }

1 + (2 + sum [3])

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[3]’ }

1 + (2 + (3 + sum []))

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[]’ }

1 + (2 + (3 + 0))

```
sum []       = 0
sum (n:ns)   = n + sum ns
```

sum [1, 2, 3]

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[1, 2, 3]’ }

1 + sum [2, 3]

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[2, 3]’ }

1 + (2 + sum [3])

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[3]’ }

1 + (2 + (3 + sum []))

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[]’ }

1 + (2 + (3 + 0))

= { *stosowanie* funkcji ‘+’ }

```
sum []       = 0
sum (n:ns)   = n + sum ns
```

sum [1, 2, 3]

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[1, 2, 3]’ }

1 + sum [2, 3]

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[2, 3]’ }

1 + (2 + sum [3])

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[3]’ }

1 + (2 + (3 + sum []))

= { *stosowanie* funkcji ‘sum’ do argumentu ‘[]’ }

1 + (2 + (3 + 0))

= { *stosowanie* funkcji ‘+’ }

6

Funkcje — przykład 4

```
qsort []          = []
qsort (x:xs)      = qsort smaller ++ [x] ++ qsort larger
                    where
                        smaller = [a | a <- xs, a <= x]
                        larger  = [b | b <- xs, b > x]
```

qsort [3, 5, 1, 4, 2]

Funkcje — przykład 4

```
qsort []          = []
qsort (x:xs)      = qsort smaller ++ [x] ++ qsort larger
                    where
                        smaller = [a | a <- xs, a <= x]
                        larger  = [b | b <- xs, b > x]
```

qsort [3, 5, 1, 4, 2]

= { *stosowanie* funkcji ‘qsort’ do argumentu ‘[3, 5, 1, 4, 2]’ }

Funkcje — przykład 4

```
qsort []          = []
qsort (x:xs)      = qsort smaller ++ [x] ++ qsort larger
                    where
                        smaller = [a | a <- xs, a <= x]
                        larger  = [b | b <- xs, b > x]
```

qsort [3, 5, 1, 4, 2]

= { *stosowanie* funkcji ‘qsort’ do argumentu ‘[3, 5, 1, 4, 2]’ }

qsort [1, 2] ++ [3] ++ qsort [5, 4]

Funkcje — przykład 4

```
qsort []          = []
qsort (x:xs)      = qsort smaller ++ [x] ++ qsort larger
                    where
                        smaller = [a | a <- xs, a <= x]
                        larger  = [b | b <- xs, b > x]
```

qsort [3, 5, 1, 4, 2]

= { *stosowanie* funkcji ‘qsort’ do argumentu ‘[3, 5, 1, 4, 2]’ }

qsort [1, 2] ++ [3] ++ qsort [5, 4]

= { *stosowanie* funkcji ‘qsort’ do argumentu ‘[1, 2]’ oraz ‘[5, 4]’ }

Funkcje — przykład 4

```
qsort []          = []
qsort (x:xs)      = qsort smaller ++ [x] ++ qsort larger
                    where
                        smaller = [a | a <- xs, a <= x]
                        larger  = [b | b <- xs, b > x]
```

qsort [3, 5, 1, 4, 2]

= { *stosowanie* funkcji ‘qsort’ do argumentu ‘[3, 5, 1, 4, 2]’ }

qsort [1, 2] ++ [3] ++ qsort [5, 4]

= { *stosowanie* funkcji ‘qsort’ do argumentu ‘[1, 2]’ oraz ‘[5, 4]’ }

(qsort [] ++ [1] ++ qsort [2]) ++ [3] ++ (qsort [4] ++ [5] ++ qsort [])

Funkcje — przykład 4

```
qsort []          = []
qsort (x:xs)      = qsort smaller ++ [x] ++ qsort larger
                    where
                        smaller = [a | a <- xs, a <= x]
                        larger  = [b | b <- xs, b > x]
```

qsort [3, 5, 1, 4, 2]

= { *stosowanie* funkcji ‘qsort’ do argumentu ‘[3, 5, 1, 4, 2]’ }

qsort [1, 2] ++ [3] ++ qsort [5, 4]

= { *stosowanie* funkcji ‘qsort’ do argumentu ‘[1, 2]’ oraz ‘[5, 4]’ }

(qsort [] ++ [1] ++ qsort [2]) ++ [3] ++ (qsort [4] ++ [5] ++ qsort [])

= { *stosowanie* funkcji ‘qsort’ do argumentu ‘[]’, ‘[2]’ oraz ‘[4]’ }

Funkcje — przykład 4

```
qsort []          = []
qsort (x:xs)      = qsort smaller ++ [x] ++ qsort larger
                    where
                        smaller = [a | a <- xs, a <= x]
                        larger  = [b | b <- xs, b > x]
```

qsort [3, 5, 1, 4, 2]

= { *stosowanie* funkcji ‘qsort’ do argumentu ‘[3, 5, 1, 4, 2]’ }

qsort [1, 2] ++ [3] ++ qsort [5, 4]

= { *stosowanie* funkcji ‘qsort’ do argumentu ‘[1, 2]’ oraz ‘[5, 4]’ }

(qsort [] ++ [1] ++ qsort [2]) ++ [3] ++ (qsort [4] ++ [5] ++ qsort [])

= { *stosowanie* funkcji ‘qsort’ do argumentu ‘[]’, ‘[2]’ oraz ‘[4]’ }

([] ++ [1] ++ [2]) ++ [3] ++ ([4] ++ [5] ++ [])

Funkcje — przykład 4

```
qsort []          = []
qsort (x:xs)      = qsort smaller ++ [x] ++ qsort larger
                    where
                        smaller = [a | a <- xs, a <= x]
                        larger  = [b | b <- xs, b > x]
```

qsort [3, 5, 1, 4, 2]

= { *stosowanie* funkcji ‘qsort’ do argumentu ‘[3, 5, 1, 4, 2]’ }

qsort [1, 2] ++ [3] ++ qsort [5, 4]

= { *stosowanie* funkcji ‘qsort’ do argumentu ‘[1, 2]’ oraz ‘[5, 4]’ }

(qsort [] ++ [1] ++ qsort [2]) ++ [3] ++ (qsort [4] ++ [5] ++ qsort [])

= { *stosowanie* funkcji ‘qsort’ do argumentu ‘[]’, ‘[2]’ oraz ‘[4]’ }

([] ++ [1] ++ [2]) ++ [3] ++ ([4] ++ [5] ++ [])

= { *stosowanie* funkcji ‘++’ do argumentów w nawiasach }

Funkcje — przykład 4

```
qsort []          = []
qsort (x:xs)      = qsort smaller ++ [x] ++ qsort larger
                    where
                        smaller = [a | a <- xs, a <= x]
                        larger  = [b | b <- xs, b > x]
```

qsort [3, 5, 1, 4, 2]

= { *stosowanie* funkcji ‘qsort’ do argumentu ‘[3, 5, 1, 4, 2]’ }

qsort [1, 2] ++ [3] ++ qsort [5, 4]

= { *stosowanie* funkcji ‘qsort’ do argumentu ‘[1, 2]’ oraz ‘[5, 4]’ }

(qsort [] ++ [1] ++ qsort [2]) ++ [3] ++ (qsort [4] ++ [5] ++ qsort [])

= { *stosowanie* funkcji ‘qsort’ do argumentu ‘[]’, ‘[2]’ oraz ‘[4]’ }

([] ++ [1] ++ [2]) ++ [3] ++ ([4] ++ [5] ++ [])

= { *stosowanie* funkcji ‘++’ do argumentów w nawiasach }

[1, 2] ++ [3] ++ [4, 5]

Funkcje — przykład 4

```
qsort []          = []
qsort (x:xs)      = qsort smaller ++ [x] ++ qsort larger
                    where
                        smaller = [a | a <- xs, a <= x]
                        larger  = [b | b <- xs, b > x]
```

qsort [3, 5, 1, 4, 2]

= { *stosowanie* funkcji ‘qsort’ do argumentu ‘[3, 5, 1, 4, 2]’ }

qsort [1, 2] ++ [3] ++ qsort [5, 4]

= { *stosowanie* funkcji ‘qsort’ do argumentu ‘[1, 2]’ oraz ‘[5, 4]’ }

(qsort [] ++ [1] ++ qsort [2]) ++ [3] ++ (qsort [4] ++ [5] ++ qsort [])

= { *stosowanie* funkcji ‘qsort’ do argumentu ‘[]’, ‘[2]’ oraz ‘[4]’ }

([] ++ [1] ++ [2]) ++ [3] ++ ([4] ++ [5] ++ [])

= { *stosowanie* funkcji ‘++’ do argumentów w nawiasach }

[1, 2] ++ [3] ++ [4, 5]

= { *stosowanie* funkcji ‘++’ }

Funkcje — przykład 4

```
qsort []          = []
qsort (x:xs)      = qsort smaller ++ [x] ++ qsort larger
                    where
                        smaller = [a | a <- xs, a <= x]
                        larger  = [b | b <- xs, b > x]
```

qsort [3, 5, 1, 4, 2]

= { *stosowanie* funkcji ‘qsort’ do argumentu ‘[3, 5, 1, 4, 2]’ }

qsort [1, 2] ++ [3] ++ qsort [5, 4]

= { *stosowanie* funkcji ‘qsort’ do argumentu ‘[1, 2]’ oraz ‘[5, 4]’ }

(qsort [] ++ [1] ++ qsort [2]) ++ [3] ++ (qsort [4] ++ [5] ++ qsort [])

= { *stosowanie* funkcji ‘qsort’ do argumentu ‘[]’, ‘[2]’ oraz ‘[4]’ }

([] ++ [1] ++ [2]) ++ [3] ++ ([4] ++ [5] ++ [])

= { *stosowanie* funkcji ‘++’ do argumentów w nawiasach }

[1, 2] ++ [3] ++ [4, 5]

= { *stosowanie* funkcji ‘++’ }

[1, 2, 3, 4, 5]

Zadania

1. Podaj inny sposób obliczenia wyrażenia: ‘double (double 2)’.
2. Wykaż, że ‘sum [x] = x’ dla dowolnej liczby ‘x’.
3. Zdefiniuj funkcję ‘product’, która mnoży wszystkie elementy z listy, np.:

```
> product [2,3,4]
  24
```

4. Jak można zmodyfikować funckję ‘qsort’ aby zwracała listę posortowaną w odwrotnej kolejności?
5. Jaki efekt będzie miała zamiana ‘<=’ na ‘<’ w definicji funkcji ‘qsort’? Rozważ przykład: ‘qsort [2, 2, 3, 1, 1]’.

Przykłady operacji na listach — 1

- Wybierz pierwszy element niepustej listy.

```
> head [1,2,3,4,5]
  1
```

Przykłady operacji na listach — 1

- Wybierz pierwszy element niepustej listy.

```
> head [1,2,3,4,5]
  1
```

- Usuń pierwszy element niepustej listy.

```
> tail [1,2,3,4,5]
  [2,3,4,5]
```

Przykłady operacji na listach — 1

- Wybierz pierwszy element niepustej listy.

```
> head [1,2,3,4,5]
  1
```

- Usuń pierwszy element niepustej listy.

```
> tail [1,2,3,4,5]
  [2,3,4,5]
```

- Wybierz n-ty elemet listy.

```
> [1,2,3,4,5] !! 2
  3
```

Przykłady operacji na listach — 1

- Wybierz pierwszy element niepustej listy.

```
> head [1,2,3,4,5]
  1
```

- Usuń pierwszy element niepustej listy.

```
> tail [1,2,3,4,5]
  [2,3,4,5]
```

- Wybierz n-ty elemet listy.

```
> [1,2,3,4,5] !! 2
  3
```

- Wybierz pierwsze n elementów z listy.

```
> take 3 [1,2,3,4,5]
  [1,2,3]
```

Przykłady operacji na listach — 2

- Usuń n pierwszych elementów z listy.

```
> drop 3 [1,2,3,4,5]
  [4,5]
```

Przykłady operacji na listach — 2

- Usuń n pierwszych elementów z listy.

```
> drop 3 [1,2,3,4,5]
  [4,5]
```

- Oblicz długość listy.

```
> length [1,2,3,4,5]
  5
```

Przykłady operacji na listach — 2

- Usuń n pierwszych elementów z listy.

```
> drop 3 [1,2,3,4,5]
  [4,5]
```

- Oblicz długość listy.

```
> length [1,2,3,4,5]
  5
```

- Oblicz sumę wszystkich elementów listy.

```
> sum [1,2,3,4,5]
  15
```

Przykłady operacji na listach — 2

- Usuń n pierwszych elementów z listy.

```
> drop 3 [1,2,3,4,5]
  [4,5]
```

- Oblicz długość listy.

```
> length [1,2,3,4,5]
  5
```

- Oblicz sumę wszystkich elementów listy.

```
> sum [1,2,3,4,5]
  15
```

- Oblicz iloczyn wszystkich elementów listy.

```
> product [1,2,3,4,5]
  120
```

Przykłady operacji na listach — 3

- Dokonaj konkatenacji dwóch list.

```
> [1,2,3] ++ [4,5]
  [1,2,3,4,5]

> [1..5] ++ []
  [1,2,3,4,5]
```

Przykłady operacji na listach — 3

- Dokonaj konkatenacji dwóch list.

```
> [1,2,3] ++ [4,5]
  [1,2,3,4,5]

> [1..5] ++ []
  [1,2,3,4,5]
```

- Odwróć kolejność elementów listy.

```
> reverse [1,2,3,4,5]
  [5,4,3,2,1]
```

Notacja matematyczna	Haskell
$f(x)$	`f x`
$f(x, y)$	`f x y`
$f(g(x))$	`f (g x)`
$f(x, g(y))$	`f x (g x)`
$f(x)g(y)$	`f x * g y`

- Dlaczego zapisując $f(g(x))$ jako `f (g x)` musimy użyć nawiasów?
- Zapisz w nowej notacji nastepującą funkcję.

$$f(x, y, g(f(u), f(z)))$$

- Zapisz w notacji matematcznej nastepującą funkcję.

```
f (g y) x z (f (g y z))
```

1. Funkcja last wybiera ostatni element niepustej listy.

```
> last [1,2,3,4,5]
  5
```

   Zdefiniuj funkcję `last` używając wprowadzonych wcześniej operacji na listach. Podaj dwie różne definicje.

2. Funkcja `init` usuwa ostatni element z niepustej listy.

```
> init [1,2,3,4,5]
  [1,2,3,4]
```

   Podaj dwie różne definicje funkcji `init`, odwołując się do wprowadzonych wcześniej operacji.

3. Spróbuj zdefiniować rekurencyjnie omówione przykładowe operacje na listach.