Czym do diaska jest programowanie funkcyjne?

Panicz Maciej Godek

godek.maciek@gmail.com

Hackerspace Trójmiasto, 07.02.2017

Definicja

Czym jest programowanie?

- stosunkowo młoda dziedzina ludzkiej działalności
  ale czy na pewno?

> *„Nauczyciele, generałowie, dietetycy, psychologowie i rodzice programują. Działania wojska, studentów i niektórych społeczności są programowane. Zmierzenie się z dużym problemem wymaga wielu programów, z których większość jest powoływana do życia w trakcie tych zmagań.”*
>
> – Alan Perlis

Definicja

Czym jest programowanie?

- stosunkowo młoda dziedzina ludzkiej działalności

  ale czy na pewno?

> *„Nauczyciele, generałowie, dietetycy, psychologowie i rodzice programują. Działania wojska, studentów i niektórych społeczności są programowane. Zmierzenie się z dużym problemem wymaga wielu programów, z których większość jest powoływana do życia w trakcie tych zmagań.”*
>
> – Alan Perlis

Definicja

Czym jest programowanie?

- stosunkowo młoda dziedzina ludzkiej działalności
  ale czy na pewno?

> *„Nauczyciele, generałowie, dietetycy, psychologowie i rodzice programują. Działania wojska, studentów i niektórych społeczności są programowane. Zmierzenie się z dużym problemem wymaga wielu programów, z których większość jest powoływana do życia w trakcie tych zmagań.”*
>
> – Alan Perlis

Definicja

Czym jest programowanie?

- stosunkowo młoda dziedzina ludzkiej działalności
  ale czy na pewno?

> *„Nauczyciele, generałowie, dietetycy, psychologowie i rodzice programują. Działania wojska, studentów i niektórych społeczności są programowane. Zmierzenie się z dużym problemem wymaga wielu programów, z których większość jest powoływana do życia w trakcie tych zmagań.”*
>
> – Alan Perlis

Analogia

Do jakiej innej dziedziny ludzkiej działalności można porównać programowanie?

- budowanie urządzeń, które przeprowadzając kolejne kroki zmieniają swój stan
- logika, wyrażanie idei (definicje, twierdzenia, dowody, przykłady)

Analogia

Do jakiej innej dziedziny ludzkiej działalności można porównać programowanie?

- budowanie urządzeń, które przeprowadzając kolejne kroki zmieniają swój stan
- logika, wyrażanie idei (definicje, twierdzenia, dowody, przykłady)

Analogia

Do jakiej innej dziedziny ludzkiej działalności można porównać programowanie?

- budowanie urządzeń, które przeprowadzając kolejne kroki zmieniają swój stan
- logika, wyrażanie idei (definicje, twierdzenia, dowody, przykłady)

Analogia

Do jakiej innej dziedziny ludzkiej działalności można porównać programowanie?

- budowanie urządzeń, które przeprowadzając kolejne kroki zmieniają swój stan
- logika, wyrażanie idei (definicje, twierdzenia, dowody, przykłady)

Analogia

Do jakiej innej dziedziny ludzkiej działalności można porównać programowanie?

- budowanie urządzeń, które przeprowadzając kolejne kroki zmieniają swój stan
- logika, wyrażanie idei (definicje, twierdzenia, dowody, przykłady)

Analogia

Do jakiej innej dziedziny ludzkiej działalności można porównać programowanie?

- budowanie urządzeń, które przeprowadzając kolejne kroki zmieniają swój stan
- logika, wyrażanie idei (definicje, twierdzenia, dowody, przykłady)

Analogia

Do jakiej innej dziedziny ludzkiej działalności można porównać programowanie?

- budowanie urządzeń, które przeprowadzając kolejne kroki ~~zmieniają swój stan~~
- logika, wyrażanie idei (definicje, twierdzenia, dowody, przykłady)

Analogia

Do jakiej innej dziedziny ludzkiej działalności można porównać programowanie?

- budowanie urządzeń~~, które przeprowadzając kolejne kroki zmieniają swój stan~~
- logika, wyrażanie idei
  (definicje, twierdzenia, dowody, przykłady)

Analogia

Do jakiej innej dziedziny ludzkiej działalności można porównać programowanie?

- ~~budowanie urządzeń, które przeprowadzając kolejne kroki zmieniają swój stan~~
- logika, wyrażanie idei
  (definicje, twierdzenia, dowody, przykłady)

Przykład

Program, który liczy sumę kwadratów początkowych siedmiu liczb pierwszych

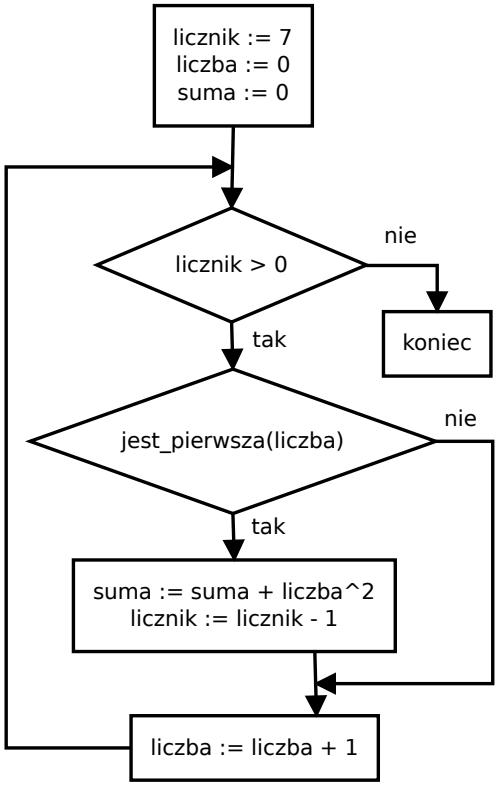
```
licznik := 7
liczba := 0
suma := 0
dopóki(licznik > 0):
    jeżeli jest_pierwsza(liczba):
        suma := suma + liczba^2
        licznik := licznik - 1
    liczba := liczba + 1
```

Przykład

Program, który liczy sumę kwadratów początkowych siedmiu liczb pierwszych

```
licznik := 7
liczba := 0
suma := 0
dopóki(licznik > 0):
    jeżeli jest_pierwsza(liczba):
        suma := suma + liczba^2
        licznik := licznik - 1
    liczba := liczba + 1
```

Schemat blokowy



Perspektywa lingwistyczna

suma kwardatów początkowych siedmiu liczb pierwszych

„suma *X*”, gdzie

X = „kwardaty *Y*”, gdzie

Y = „*k* początkowych liczb pierwszych”, gdzie

k = 7

Perspektywa lingwistyczna

suma kwardatów początkowych siedmiu liczb pierwszych
„suma *X*”, gdzie
X = „kwardaty *Y*”, gdzie
Y = „*k* początkowych liczb pierwszych”, gdzie
k = 7

Perspektywa lingwistyczna

suma kwardatów początkowych siedmiu liczb pierwszych
„suma *X*”, gdzie
X = „kwardaty *Y*”, gdzie
Y = „*k* początkowych liczb pierwszych”, gdzie
k = 7

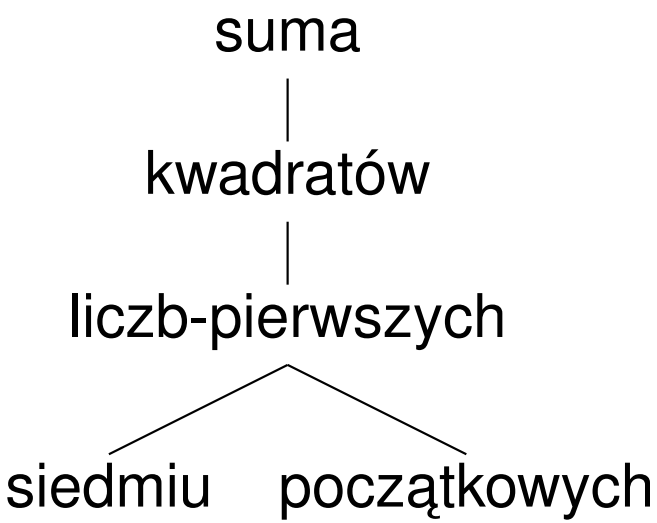
Perspektywa lingwistyczna

suma kwardatów początkowych siedmiu liczb pierwszych
„suma *X*”, gdzie
X = „kwardaty *Y*”, gdzie
Y = „*k* początkowych liczb pierwszych”, gdzie
k = 7

Perspektywa lingwistyczna

suma kwardatów początkowych siedmiu liczb pierwszych
„suma *X*”, gdzie
X = „kwardaty *Y*”, gdzie
Y = „*k* początkowych liczb pierwszych”, gdzie
k = 7

Drzewko



„suma kwadratów początkowych siedmiu liczb pierwszych”
Do wyjaśnienia:

- co znaczy „suma *X*”?
- co znaczy „kwadraty *Y*”
- co znaczy „siedem początkowych liczb pierwszych”?

„suma kwadratów początkowych siedmiu liczb pierwszych”
Do wyjaśnienia:

- co znaczy „suma *X*”?
- co znaczy „kwadraty *Y*”
- co znaczy „siedem początkowych liczb pierwszych”?

„suma kwadratów początkowych siedmiu liczb pierwszych”
Do wyjaśnienia:

- co znaczy „suma *X*”?
- co znaczy „kwadraty *Y*”
- co znaczy „siedem początkowych liczb pierwszych”?

„suma kwadratów początkowych siedmiu liczb pierwszych”
Do wyjaśnienia:

- co znaczy „suma *X*”?
- co znaczy „kwadraty *Y*”
- co znaczy „siedem początkowych liczb pierwszych”?

Znaczenie

Co znaczy „siedem początkowych liczb pierwszych”?

denotacja vs. konotacja

- denotacja (odniesienie): obiekt, do którego dane wyrażenie językowe się *odnosi*, np. ciąg `(2 3 5 7 11 13 17)`
- konotacja (określenie): inne wyrażenie językowe równoważne danemu (ale prostsze pojęciowo)

Znaczenie

Co znaczy „siedem początkowych liczb pierwszych”?

denotacja vs. konotacja

- denotacja (odniesienie): obiekt, do którego dane wyrażenie językowe się *odnosi*, np. ciąg `(2 3 5 7 11 13 17)`
- konotacja (określenie): inne wyrażenie językowe równoważne danemu (ale prostsze pojęciowo)

Znaczenie

Co znaczy „siedem początkowych liczb pierwszych”?

denotacja vs. konotacja

- denotacja (odniesienie): obiekt, do którego dane wyrażenie językowe się *odnosi*, np. ciąg `(2 3 5 7 11 13 17)`
- konotacja (określenie): inne wyrażenie językowe równoważne danemu (ale prostsze pojęciowo)

Znaczenie

Co znaczy „siedem początkowych liczb pierwszych”?

denotacja vs. konotacja

- denotacja (odniesienie): obiekt, do którego dane wyrażenie językowe się *odnosi*, np. ciąg `(2 3 5 7 11 13 17)`
- konotacja (określenie): inne wyrażenie językowe równoważne danemu (ale prostsze pojęciowo)

Znaczenie

Co znaczy „siedem początkowych liczb pierwszych”?

denotacja vs. konotacja

- denotacja (odniesienie): obiekt, do którego dane wyrażenie językowe się *odnosi*, np. ciąg `(2 3 5 7 11 13 17)`
- konotacja (określenie): inne wyrażenie językowe równoważne danemu (ale prostsze pojęciowo)

Redukcja

Konotacja (definicja)
Niech *N* i *M* oznaczają liczby naturalne. Rozważmy znaczenie wyrażenia
„*N* najmniejszych liczb pierwszych większych od *M*”

1. wiemy, że znaczeniem tego wyrażenia (o ile jest sensowne) jest **ciąg** *N*-elementowy
2. znaczenie wyrażenia „0 najmniejszych liczb pierwszych większych od *M*” jest **tożsame** (koekstensjonalne) ze znaczeniem wyrażenia „pusty ciąg”

Redukcja

Konotacja (definicja)
Niech *N* i *M* oznaczają liczby naturalne. Rozważmy znaczenie wyrażenia
„*N* najmniejszych liczb pierwszych większych od *M*”

1. wiemy, że znaczeniem tego wyrażenia (o ile jest sensowne) jest **ciąg** *N*-elementowy
2. znaczenie wyrażenia „0 najmniejszych liczb pierwszych większych od *M*” jest **tożsame** (koekstensjonalne) ze znaczeniem wyrażenia „pusty ciąg”

Redukcja

Konotacja (definicja)
Niech *N* i *M* oznaczają liczby naturalne. Rozważmy znaczenie wyrażenia
„*N* najmniejszych liczb pierwszych większych od *M*”

1. wiemy, że znaczeniem tego wyrażenia (o ile jest sensowne) jest **ciąg** *N*-elementowy
2. znaczenie wyrażenia „0 najmniejszych liczb pierwszych większych od *M*” jest **tożsame** (koekstensjonalne) ze znaczeniem wyrażenia „pusty ciąg”

Redukcja

Konotacja (definicja) Niech N i M oznaczają liczby naturalne. Rozważmy znaczenie wyrażenia
„$N + 1$ najmniejszych liczb pierwszych większych od M”

3. jeżeli $M + 1$ jest liczbą pierwszą, to znaczeniem wyrażenia „$N + 1$ najmniejszych liczb pierwszych większych od M” jest **ciąg**, którego pierwszym elementem jest $M + 1$, a którego pozostałe elementy to „N najmniejszych liczb pierwszych większych od $M + 1$
4. jeżeli $M + 1$ nie jest liczbą pierwszą, to znaczenie wyrażenia „$N + 1$ najmniejszych liczb pierwszych większych od M” jest **takie samo**, jak znaczenie wyrażenia „$N + 1$ liczb pierwszych większych od $M + 1$”.

Redukcja

Konotacja (definicja) Niech N i M oznaczają liczby naturalne. Rozważmy znaczenie wyrażenia
„$N + 1$ najmniejszych liczb pierwszych większych od M”

3. jeżeli $M + 1$ jest liczbą pierwszą, to znaczeniem wyrażenia „$N + 1$ najmniejszych liczb pierwszych większych od M” jest **ciąg**, którego pierwszym elementem jest $M + 1$, a którego pozostałe elementy to „N najmniejszych liczb pierwszych większych od $M + 1$
4. jeżeli $M + 1$ nie jest liczbą pierwszą, to znaczenie wyrażenia „$N + 1$ najmniejszych liczb pierwszych większych od M” jest **takie samo**, jak znaczenie wyrażenia „$N + 1$ liczb pierwszych większych od $M + 1$”.

Redukcja

Konotacja (definicja) Niech N i M oznaczają liczby naturalne. Rozważmy znaczenie wyrażenia
„$N + 1$ najmniejszych liczb pierwszych większych od M”

3. jeżeli $M + 1$ jest liczbą pierwszą, to znaczeniem wyrażenia „$N + 1$ najmniejszych liczb pierwszych większych od M” jest **ciąg**, którego pierwszym elementem jest $M + 1$, a którego pozostałe elementy to „N najmniejszych liczb pierwszych większych od $M + 1$
4. jeżeli $M + 1$ nie jest liczbą pierwszą, to znaczenie wyrażenia „$N + 1$ najmniejszych liczb pierwszych większych od M” jest **takie samo**, jak znaczenie wyrażenia „$N + 1$ liczb pierwszych większych od $M + 1$”.

Uff...

Formalna notacja

suma kwadratów początkowych siedmiu liczb pierwszych

sum of squares of initial seven prime numbers (angielski nie ma deklinacji)

```
(sum (squares (initial seven prime numbers)))
```

(złożone deskrypcje bierzemy w nawiasy, `f of x = (f x)`)

```
(sum (squares (prime-numbers seven initial)))
```

(słowa rządzące na początku)

Formalna notacja

suma kwadratów początkowych siedmiu liczb pierwszych

sum of squares of initial seven prime numbers (angielski nie ma deklinacji)

```
(sum (squares (initial seven prime numbers)))
```

(złożone deskrypcje bierzemy w nawiasy, `f of x = (f x)`)

```
(sum (squares (prime-numbers seven initial)))
```

(słowa rządzące na początku)

Formalna notacja

suma kwadratów początkowych siedmiu liczb pierwszych

sum of squares of initial seven prime numbers (angielski nie ma deklinacji)

```
(sum (squares (initial seven prime numbers)))
```

(złożone deskrypcje bierzemy w nawiasy, `f` of `x` = `(f x)`)

```
(sum (squares (prime-numbers seven initial)))
```

(słowa rządzące na początku)

Formalna notacja

suma kwadratów początkowych siedmiu liczb pierwszych

sum of squares of initial seven prime numbers (angielski nie ma deklinacji)

```
(sum (squares (initial seven prime numbers)))
```

(złożone deskrypcje bierzemy w nawiasy, `f` of `x` = `(f x)`)

```
(sum (squares (prime-numbers seven initial)))
```

(słowa rządzące na początku)

Język Scheme

```
(define (prime-numbers amount lowest)
 (if (equal? amount 0)
   '()
   (if (prime? lowest)
     (cons lowest (prime-numbers (- amount 1)
                                 (+ lowest 1)))
     (prime-numbers amount (+ lowest 1)))))
```

Język Scheme

```
(define (prime-numbers amount lowest)
 (if (equal? amount 0)
   '()
   (if (prime? lowest)
     (cons lowest (prime-numbers (- amount 1)
                                 (+ lowest 1)))
     (prime-numbers amount (+ lowest 1)))))
```

Język Scheme

```
(define (prime-numbers amount lowest)
 (if (equal? amount 0)
   '()
   (if (prime? lowest)
     (cons lowest (prime-numbers (- amount 1)
                                 (+ lowest 1)))
     (prime-numbers amount (+ lowest 1)))))
```

Język Scheme

```
(define (prime-numbers amount lowest)
 (if (equal? amount 0)
   '()
   (if (prime? lowest)
     (cons lowest (prime-numbers (- amount 1)
                                 (+ lowest 1)))
     (prime-numbers amount (+ lowest 1)))))
```

Język Scheme

```
(define (prime-numbers amount lowest)
 (if (equal? amount 0)
   '()
   (if (prime? lowest)
     (cons lowest (prime-numbers (- amount 1)
                                 (+ lowest 1)))
     (prime-numbers amount (+ lowest 1)))))
```

Język Scheme

```
(define (prime-numbers amount lowest)
 (if (equal? amount 0)
   '()
   (if (prime? lowest)
     (cons lowest (prime-numbers (- amount 1)
                                 (+ lowest 1)))
     (prime-numbers amount (+ lowest 1)))))
```

Język Scheme

```
(define (prime-numbers amount lowest)
 (if (equal? amount 0)
   '()
   (if (prime? lowest)
     (cons lowest (prime-numbers (- amount 1)
                                 (+ lowest 1)))
     (prime-numbers amount (+ lowest 1)))))
```

Pojęcia pierwotne

Użyte pojęcia pierwotne:

- `(if <warunek> <wartość> <alternatywa>)` – jeżeli `<warunek>` jest spełniony, znaczeniem całego wyrażenia jest `<wartość>`, a w przeciwnym razie jest nią `<alternatywa>`
- `(equal? a b)` – `a` i `b` są równe
- `(+ a b)` – suma `a` i `b`
- `(- a b)` – różnica `a` i `b`
- `(cons element sequence)` – sekwencja, której pierwszym elementem jest `element`, zaś pozostałe elementy to elementy sekwencji `sequence`
- `'()` – sekwencja pusta

Pojęcia pierwotne

Użyte pojęcia pierwotne:

- `(if <warunek> <wartość> <alternatywa>)` – jeżeli `<warunek>` jest spełniony, znaczeniem całego wyrażenia jest `<wartość>`, a w przeciwnym razie jest nią `<alternatywa>`
- `(equal? a b)` – `a` i `b` są równe
- `(+ a b)` – suma `a` i `b`
- `(- a b)` – różnica `a` i `b`
- `(cons element sequence)` – sekwencja, której pierwszym elementem jest `element`, zaś pozostałe elementy to elementy sekwencji `sequence`
- `'()` – sekwencja pusta

Pojęcia pierwotne

Użyte pojęcia pierwotne:

- `(if <warunek> <wartość> <alternatywa>)` – jeżeli `<warunek>` jest spełniony, znaczeniem całego wyrażenia jest `<wartość>`, a w przeciwnym razie jest nią `<alternatywa>`
- `(equal? a b)` – `a` i `b` są równe
- `(+ a b)` – suma `a` i `b`
- `(- a b)` – różnica `a` i `b`
- `(cons element sequence)` – sekwencja, której pierwszym elementem jest `element`, zaś pozostałe elementy to elementy sekwencji `sequence`
- `'()` – sekwencja pusta

Pojęcia pierwotne

Użyte pojęcia pierwotne:

- `(if <warunek> <wartość> <alternatywa>)` – jeżeli `<warunek>` jest spełniony, znaczeniem całego wyrażenia jest `<wartość>`, a w przeciwnym razie jest nią `<alternatywa>`
- `(equal? a b)` – `a` i `b` są równe
- `(+ a b)` – suma `a` i `b`
- `(- a b)` – różnica `a` i `b`
- `(cons element sequence)` – sekwencja, której pierwszym elementem jest `element`, zaś pozostałe elementy to elementy sekwencji `sequence`
- `'()` – sekwencja pusta

Pojęcia pierwotne

Użyte pojęcia pierwotne:

- `(if <warunek> <wartość> <alternatywa>)` – jeżeli `<warunek>` jest spełniony, znaczeniem całego wyrażenia jest `<wartość>`, a w przeciwnym razie jest nią `<alternatywa>`
- `(equal? a b)` – `a` i `b` są równe
- `(+ a b)` – suma `a` i `b`
- `(- a b)` – różnica `a` i `b`
- `(cons element sequence)` – sekwencja, której pierwszym elementem jest `element`, zaś pozostałe elementy to elementy sekwencji `sequence`
- `'()` – sekwencja pusta

Pojęcia pierwotne

Użyte pojęcia pierwotne:

- `(if <warunek> <wartość> <alternatywa>)` – jeżeli `<warunek>` jest spełniony, znaczeniem całego wyrażenia jest `<wartość>`, a w przeciwnym razie jest nią `<alternatywa>`
- `(equal? a b)` – `a` i `b` są równe
- `(+ a b)` – suma `a` i `b`
- `(- a b)` – różnica `a` i `b`
- `(cons element sequence)` – sekwencja, której pierwszym elementem jest `element`, zaś pozostałe elementy to elementy sekwencji `sequence`
- `'()` – sekwencja pusta

Pojęcia pierwotne

Użyte pojęcia pierwotne:

- `(if <warunek> <wartość> <alternatywa>)` – jeżeli `<warunek>` jest spełniony, znaczeniem całego wyrażenia jest `<wartość>`, a w przeciwnym razie jest nią `<alternatywa>`
- `(equal? a b)` – `a` i `b` są równe
- `(+ a b)` – suma `a` i `b`
- `(- a b)` – różnica `a` i `b`
- `(cons element sequence)` – sekwencja, której pierwszym elementem jest `element`, zaś pozostałe elementy to elementy sekwencji `sequence`
- `'()` – sekwencja pusta

Pojęcia wtórne

Użyte pojęcia wtórne:

- `prime-numbers` – ale to je właśnie definiujemy (rekurencyjnie)
- `(prime? n)` – test, czy `n` jest liczbą pierwszą

Pojęcia wtórne

Użyte pojęcia wtórne:

- `prime-numbers` – ale to je właśnie definiujemy (rekurencyjnie)
- `(prime? n)` – test, czy `n` jest liczbą pierwszą

Pojęcia wtórne

Użyte pojęcia wtórne:

- `prime-numbers` – ale to je właśnie definiujemy (rekurencyjnie)
- `(prime? n)` – test, czy `n` jest liczbą pierwszą

Pierwszość liczby

n jest *liczbą pierwszą*, jeśli jej jedyne podzielniki to 1 oraz *n*

```
(define (prime? n)
  (equal?  (divisors n) (list 1 n)))
```

pojęcia pierwotne:

- `(list arg1 arg2 ...)` – lista (sekwencja) zawierająca argumenty `arg1`, `arg2`, `...`

pojęcia wtórne:

- `(divisors n)` – lista wszystkich podzielników liczby naturalnej `n`

Pierwszość liczby

n jest *liczbą pierwszą*, jeśli jej jedyne podzielniki to 1 oraz *n*

```
(define (prime? n)
  (equal?  (divisors n) (list 1 n)))
```

pojęcia pierwotne:

- `(list arg1 arg2 ...)` – lista (sekwencja) zawierająca argumenty `arg1`, `arg2`, `...`

pojęcia wtórne:

- `(divisors n)` – lista wszystkich podzielników liczby naturalnej `n`

Pierwszość liczby

n jest *liczbą pierwszą*, jeśli jej jedyne podzielniki to 1 oraz *n*

```
(define (prime? n)
  (equal?  (divisors n) (list 1 n)))
```

pojęcia pierwotne:

- `(list arg1 arg2 ...)` – lista (sekwencja) zawierająca argumenty `arg1`, `arg2`, `...`

pojęcia wtórne:

- `(divisors n)` – lista wszystkich podzielników liczby naturalnej `n`

Pierwszość liczby

n jest *liczbą pierwszą*, jeśli jej jedyne podzielniki to 1 oraz *n*

```
(define (prime? n)
  (equal?  (divisors n) (list 1 n)))
```

pojęcia pierwotne:

- `(list arg1 arg2 ...)` – lista (sekwencja) zawierająca argumenty `arg1`, `arg2`, `...`

pojęcia wtórne:

- `(divisors n)` – lista wszystkich podzielników liczby naturalnej `n`

Pierwszość liczby

n jest *liczbą pierwszą*, jeśli jej jedyne podzielniki to 1 oraz *n*

```
(define (prime? n)
  (equal?  (divisors n) (list 1 n)))
```

pojęcia pierwotne:

- `(list arg1 arg2 ...)` – lista (sekwencja) zawierająca argumenty `arg1`, `arg2`, `...`

pojęcia wtórne:

- `(divisors n)` – lista wszystkich podzielników liczby naturalnej `n`

Pierwszość liczby

n jest *liczbą pierwszą*, jeśli jej jedyne podzielniki to 1 oraz *n*

```
(define (prime? n)
  (equal?  (divisors n) (list 1 n)))
```

pojęcia pierwotne:

- `(list arg1 arg2 ...)` – lista (sekwencja) zawierająca argumenty `arg1`, `arg2`, `...`

pojęcia wtórne:

- `(divisors n)` – lista wszystkich podzielników liczby naturalnej `n`

Podzielniki liczby

Liczba $1 \leq k \leq n$ jest podzielnikiem liczby n wtedy i tylko wtedy, gdy reszta z dzielenia n przez k wynosi zero

```
(define (divisors n)
 (define (divisors n k)
   (if (> k n)
     '()
     (if (equal? (remainder n k) 0)
       (cons k (divisors n (+ k 1)))
       (divisors n (+ k 1)))))
 (divisors n 1))
```

Podzielniki liczby

Liczba $1 \leq k \leq n$ jest podzielnikiem liczby n wtedy i tylko wtedy, gdy reszta z dzielenia n przez k wynosi zero

```
(define (divisors n)
 (define (divisors n k)
   (if (> k n)
     '()
     (if (equal? (remainder n k) 0)
       (cons k (divisors n (+ k 1)))
       (divisors n (+ k 1)))))
 (divisors n 1))
```

Podzielniki liczby

Liczba $1 \leq k \leq n$ jest podzielnikiem liczby n wtedy i tylko wtedy, gdy reszta z dzielenia n przez k wynosi zero

```
(define (divisors n)
 (define (divisors n k)
   (if (> k n)
     '()
     (if (equal? (remainder n k) 0)
       (cons k (divisors n (+ k 1)))
       (divisors n (+ k 1)))))
 (divisors n 1)
```

Podzielniki liczby

Liczba $1 \leq k \leq n$ jest podzielnikiem liczby n wtedy i tylko wtedy, gdy reszta z dzielenia n przez k wynosi zero

```
(define (divisors n)
 (define (divisors n k)
   (if (> k n)
     '()
     (if (equal? (remainder n k) 0)
       (cons k (divisors n (+ k 1)))
       (divisors n (+ k 1)))))
 (divisors n 1))
```

Podzielniki liczby

Liczba $1 \leq k \leq n$ jest podzielnikiem liczby n wtedy i tylko wtedy, gdy reszta z dzielenia n przez k wynosi zero

```
(define (divisors n)
 (define (divisors n k)
   (if (> k n)
     '()
     (if (equal? (remainder n k) 0)
       (cons k (divisors n (+ k 1)))
       (divisors n (+ k 1)))))
 (divisors n 1))
```

Podzielniki liczby

Liczba $1 \leq k \leq n$ jest podzielnikiem liczby n wtedy i tylko wtedy, gdy reszta z dzielenia n przez k wynosi zero

```
(define (divisors n)
 (define (divisors n k)
   (if (> k n)
     '()
     (if (equal? (remainder n k) 0)
       (cons k (divisors n (+ k 1)))
       (divisors n (+ k 1)))))
 (divisors n 1))
```

Podzielniki liczby

Liczba $1 \leq k \leq n$ jest podzielnikiem liczby n wtedy i tylko wtedy, gdy reszta z dzielenia n przez k wynosi zero

```
(define (divisors n)
 (define (divisors n k)
   (if (> k n)
     '()
     (if (equal? (remainder n k) 0)
       (cons k (divisors n (+ k 1)))
       (divisors n (+ k 1)))))
 (divisors n 1))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (if (> 1 4)
  '()
  (if (equal? (remainder 4 1) 0)
    (cons 1 (divisors 4 (+ 1 1)))
    (divisors 4 (+ 1 1))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (if (> 1 4)
  '()
  (if (equal? (remainder 4 1) 0)
    (cons 1 (divisors 4 (+ 1 1)))
    (divisors 4 (+ 1 1))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (if (> 1 4)
  '()
  (if (equal? (remainder 4 1) 0)
    (cons 1 (divisors 4 (+ 1 1)))
    (divisors 4 (+ 1 1))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (if (> 1 4)
  '()
  (if (equal? (remainder 4 1) 0)
    (cons 1 (divisors 4 (+ 1 1)))
    (divisors 4 (+ 1 1))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (if #false
  '()
  (if (equal? (remainder 4 1) 0)
    (cons 1 (divisors 4 (+ 1 1)))
    (divisors 4 (+ 1 1))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

`(divisors 4 1)`

```
 (if (equal? (remainder 4 1) 0)
  (cons 1 (divisors 4 (+ 1 1)))
  (divisors 4 (+ 1 1)))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (if (equal? (remainder 4 1) 0)
  (cons 1 (divisors 4 (+ 1 1)))
  (divisors 4 (+ 1 1)))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (if (equal? (remainder 4 1) 0)
  (cons 1 (divisors 4 (+ 1 1)))
  (divisors 4 (+ 1 1)))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (if (equal? 0 0)
  (cons 1 (divisors 4 (+ 1 1)))
  (divisors 4 (+ 1 1)))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (if (equal? 0 0)
  (cons 1 (divisors 4 (+ 1 1)))
  (divisors 4 (+ 1 1)))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (if #true
  (cons 1 (divisors 4 (+ 1 1)))
  (divisors 4 (+ 1 1)))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1 (divisors 4 (+ 1 1)))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1 (divisors 4 (+ 1 1)))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1 (divisors 4 2))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1 (divisors 4 2))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1 (if (> 2 4)
             '()
             (if (equal? (remainder 4 2) 0)
               (cons 2 (divisors 4 (+ 2 1)))
               (divisors 4 (+ 2 1)))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1 (if (> 2 4)
             '()
             (if (equal? (remainder 4 2) 0)
               (cons 2 (divisors 4 (+ 2 1)))
               (divisors 4 (+ 2 1))))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1 (if #false
             '()
             (if (equal? (remainder 4 2) 0)
               (cons 2 (divisors 4 (+ 2 1)))
               (divisors 4 (+ 2 1))))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1 (if (equal? (remainder 4 2) 0)
             (cons 2 (divisors 4 (+ 2 1)))
             (divisors 4 (+ 2 1))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1 (if (equal? (remainder 4 2) 0)
             (cons 2 (divisors 4 (+ 2 1)))
             (divisors 4 (+ 2 1)))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1 (if (equal? (remainder 4 2) 0)
             (cons 2 (divisors 4 (+ 2 1)))
             (divisors 4 (+ 2 1)))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1 (if (equal? 0 0)
             (cons 2 (divisors 4 (+ 2 1)))
             (divisors 4 (+ 2 1)))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1 (if (equal? 0 0)
             (cons 2 (divisors 4 (+ 2 1)))
             (divisors 4 (+ 2 1)))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1 (if #true
             (cons 2 (divisors 4 (+ 2 1)))
             (divisors 4 (+ 2 1)))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1 (cons 2 (divisors 4 (+ 2 1))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1 (cons 2 (divisors 4 (+ 2 1))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1 (cons 2 (divisors 4 3)))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (divisors 4 3)))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (if (> 3 4)
      '()
      (if (equal? (remainder 4 3) 0)
        (cons 3 (divisors 4 (+ 4 1)))
        (divisors 4 (+ 3 1))))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (if (> 3 4)
      '()
      (if (equal? (remainder 4 3) 0)
        (cons 3 (divisors 4 (+ 4 1)))
        (divisors 4 (+ 3 1))))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (if #false
      '()
      (if (equal? (remainder 4 3) 0)
        (cons 3 (divisors 4 (+ 4 1)))
        (divisors 4 (+ 3 1))))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (if (equal? (remainder 4 3) 0)
      (cons 3 (divisors 4 (+ 4 1)))
      (divisors 4 (+ 3 1)))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (if (equal? (remainder 4 3) 0)
      (cons 3 (divisors 4 (+ 4 1)))
      (divisors 4 (+ 3 1))))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (if (equal? (remainder 4 3) 0)
      (cons 3 (divisors 4 (+ 4 1)))
      (divisors 4 (+ 3 1))))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (if (equal? (remainder 4 3) 0)
      (cons 3 (divisors 4 (+ 4 1)))
      (divisors 4 (+ 3 1))))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (if (equal? 1 0)
      (cons 3 (divisors 4 (+ 4 1)))
      (divisors 4 (+ 3 1))))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (if (equal? 1 0)
      (cons 3 (divisors 4 (+ 4 1)))
      (divisors 4 (+ 3 1))))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (if #false
      (cons 3 (divisors 4 (+ 4 1)))
      (divisors 4 (+ 3 1)))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (divisors 4 (+ 3 1))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (divisors 4 (+ 3 1)))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (divisors 4 4)))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (divisors 4 4)))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (if (> 4 4)
      '()
      (if (equal? (remainder 4 4) 0)
        (cons 4 (divisors 4 (+ 4 1)))
        (divisors 4 (+ 4 1))))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (if (> 4 4)
      '()
      (if (equal? (remainder 4 4) 0)
        (cons 4 (divisors 4 (+ 4 1)))
        (divisors 4 (+ 4 1))))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (if #false
      '()
      (if (equal? (remainder 4 4) 0)
        (cons 4 (divisors 4 (+ 4 1)))
        (divisors 4 (+ 4 1))))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (if (equal? (remainder 4 4) 0)
      (cons 4 (divisors 4 (+ 4 1)))
      (divisors 4 (+ 4 1))))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (if (equal? (remainder 4 4) 0)
      (cons 4 (divisors 4 (+ 4 1)))
      (divisors 4 (+ 4 1))))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (if (equal? 0 0)
      (cons 4 (divisors 4 (+ 4 1)))
      (divisors 4 (+ 4 1))))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (if (equal? 0 0)
      (cons 4 (divisors 4 (+ 4 1)))
      (divisors 4 (+ 4 1))))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (if #true
      (cons 4 (divisors 4 (+ 4 1)))
      (divisors 4 (+ 4 1))))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (cons 4 (divisors 4 (+ 4 1)))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (cons 4 (divisors 4 (+ 4 1)))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (cons 4 (divisors 4 5))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (cons 4 (divisors 4 5))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (cons 4 (if (> 5 4)
      '()
      (if (equal? (remainder 4 5) 0)
        (cons 5 (divisors 4 (+ 5 1)))
        (divisors 5 (+ 4 1)))))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (cons 4 (if (> 5 4)
      '()
      (if (equal? (remainder 4 5) 0)
        (cons 5 (divisors 4 (+ 5 1)))
        (divisors 5 (+ 4 1)))))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1
  (cons 2
    (cons 4 (if #true
      '()
      (if (equal? (remainder 4 5) 0)
        (cons 5 (divisors 4 (+ 5 1)))
        (divisors 5 (+ 4 1)))))))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1 (cons 2 (cons 4 '())))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1 (cons 2 (cons 4 '())))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1 (cons 2 '(4)))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1 (cons 2 '(4)))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1 '(2 4))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (cons 1 '(2 4))
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 '(1 2 4)
```

Analiza

Rozważmy `(divisors 4)`. Na mocy definicji mamy

```
(divisors 4 1)
 (1 2 4)
```

```
(sum (squares (prime-numbers 7 0)))
```

Do wyjaśnienia:

- co znaczy „suma *X*”?
- co znaczy „kwadraty *Y*”
- ~~co znaczy „siedem początkowych liczb pierwszych”?~~

Kwadrat liczby

kwadratem liczby nazwiemy tę liczbę pomnożoną przez siebie samą

```
(define (square x)
  (* x x))
```

jak należy rozumieć *kwadraty ciągu liczb*?

```
(sum (squares (prime-numbers 7 0)))
```

czym jest `squares`?

Kwadrat liczby

kwadratem liczby nazwiemy tę liczbę pomnożoną przez siebie samą

```
(define (square x)
 (* x x))
```

jak należy rozumieć *kwadraty ciągu liczb*?

```
(sum (squares (prime-numbers 7 0)))
```

czym jest `squares`?

Kwadrat liczby

kwadratem liczby nazwiemy tę liczbę pomnożoną przez siebie samą

```
(define (square x)
 (* x x))
```

jak należy rozumieć *kwadraty ciągu liczb*?

```
(sum (squares (prime-numbers 7 0)))
```

czym jest `squares`?

Kwadrat liczby

kwadratem liczby nazwiemy tę liczbę pomnożoną przez siebie samą

```
(define (square x)
 (* x x))
```

jak należy rozumieć *kwadraty ciągu liczb*?

```
(sum (squares (prime-numbers 7 0)))
```

czym jest `squares`?

Kwadrat liczby

kwadratem liczby nazwiemy tę liczbę pomnożoną przez siebie samą

```
(define (square x)
 (* x x))
```

jak należy rozumieć *kwadraty ciągu liczb*?

```
(sum (squares (prime-numbers 7 0)))
```

czym jest `squares`?

Kwadraty ciągu

1. wiemy, że dowolny ciąg może być albo *ciągiem pustym*, albo posiada *pierwszy element* oraz *pozostałe elementy*
2. kwadraty pustego ciągu to ciąg pusty
3. kwadraty ciągu złożonego z pierwszego elementu oraz pozostałych elementów to ciąg, którego pierwszym elementem jest kwadrat pierwszego elementu oryginalnego ciągu, a którego pozostałe elementy to kwadraty pozostałch elementów oryginalnego ciągu

```
(define (squares sequence)
 (if (equal? sequence '())
   '()
   (cons (square (car sequence))
         (squares (cdr numbers)))))
```

Kwadraty ciągu

1. wiemy, że dowolny ciąg może być albo *ciągiem pustym*, albo posiada *pierwszy element* oraz *pozostałe elementy*
2. kwadraty pustego ciągu to ciąg pusty
3. kwadraty ciągu złożonego z pierwszego elementu oraz pozostałych elementów to ciąg, którego pierwszym elementem jest kwadrat pierwszego elementu oryginalnego ciągu, a którego pozostałe elementy to kwadraty pozostałch elementów oryginalnego ciągu

```
(define (squares sequence)
 (if (equal? sequence '())
   '()
   (cons (square (car sequence))
         (squares (cdr numbers)))))
```

Kwadraty ciągu

1. wiemy, że dowolny ciąg może być albo *ciągiem pustym*, albo posiada *pierwszy element* oraz *pozostałe elementy*
2. kwadraty pustego ciągu to ciąg pusty
3. kwadraty ciągu złożonego z pierwszego elementu oraz pozostałych elementów to ciąg, którego pierwszym elementem jest kwadrat pierwszego elementu oryginalnego ciągu, a którego pozostałe elementy to kwadraty pozostałch elementów oryginalnego ciągu

```
(define (squares sequence)
 (if (equal? sequence '())
   '()
   (cons (square (car sequence))
         (squares (cdr numbers)))))
```

Kwadraty ciągu

1. wiemy, że dowolny ciąg może być albo *ciągiem pustym*, albo posiada *pierwszy element* oraz *pozostałe elementy*
2. kwadraty pustego ciągu to ciąg pusty
3. kwadraty ciągu złożonego z pierwszego elementu oraz pozostałych elementów to ciąg, którego pierwszym elementem jest kwadrat pierwszego elementu oryginalnego ciągu, a którego pozostałe elementy to kwadraty pozostałch elementów oryginalnego ciągu

```
(define (squares sequence)
 (if (equal? sequence '())
   '()
   (cons (square (car sequence))
         (squares (cdr numbers)))))
```

Liczba mnoga

gdybyśmy chcieli policzyć sumę sześcianów jakiegoś ciągu...

musielibyśmy definiować funkcje `cube` oraz `cubes`?

przecież w języku dysponujemy *liczbą mnogą*

```
(define (map attribute sequence)
  (if (equal?  sequence '())
      '()
      (cons (attribute (car sequence))
            (map attribute (cdr sequence)))))
(sum (map square (prime-numbers 7 0)))
```

Liczba mnoga

gdybyśmy chcieli policzyć sumę sześcianów jakiegoś ciągu...
musielibyśmy definiować funkcje `cube` oraz `cubes`?
przecież w języku dysponujemy *liczbą mnogą*

```
(define (map attribute sequence)
  (if (equal?  sequence '())
      '()
      (cons (attribute (car sequence))
            (map attribute (cdr sequence)))))
(sum (map square (prime-numbers 7 0)))
```

Liczba mnoga

gdybyśmy chcieli policzyć sumę sześcianów jakiegoś ciągu...
musielibyśmy definiować funkcje `cube` oraz `cubes`?
przecież w języku dysponujemy *liczbą mnogą*

```
(define (map attribute sequence)
  (if (equal?  sequence '())
      '()
      (cons (attribute (car sequence))
            (map attribute (cdr sequence)))))
(sum (map square (prime-numbers 7 0)))
```

Liczba mnoga

gdybyśmy chcieli policzyć sumę sześcianów jakiegoś ciągu...
musielibyśmy definiować funkcje `cube` oraz `cubes`?
przecież w języku dysponujemy *liczbą mnogą*

```
(define (map attribute sequence)
 (if (equal?  sequence '())
   '()
   (cons (attribute (car sequence))
         (map attribute (cdr sequence)))))
(sum (map square (prime-numbers 7 0)))
```

Liczba mnoga

gdybyśmy chcieli policzyć sumę sześcianów jakiegoś ciągu...
musielibyśmy definiować funkcje `cube` oraz `cubes`?
przecież w języku dysponujemy *liczbą mnogą*

```
 (define (map attribute sequence)
  (if (equal?  sequence '())
    '()
    (cons (attribute (car sequence))
          (map attribute (cdr sequence)))))
(sum (map square (prime-numbers 7 0)))
```

```
(sum (map square (prime-numbers 7 0)))
```

Do wyjaśnienia:

- co znaczy „suma *X*”?
- ~~co znaczy „kwadraty *Y*”~~
- ~~co znaczy „siedem początkowych liczb pierwszych”?~~

Suma

```
(define (sum sequence)
  (if (equal?  sequence '())
      0
      (+ (car sequence)
         (sum (cdr sequence)))))
```

Suma

```
(define (sum sequence)
 (if (equal?  sequence '())
   0
   (+ (car sequence)
     (sum (cdr sequence)))))
```

Suma

```
(define (sum sequence)
 (if (equal?  sequence '())
   0
    (+ (car sequence)
      (sum (cdr sequence)))))
```

Suma

```
(define (sum sequence)
 (if (equal?  sequence '())
   0
   (+ (car sequence)
     (sum (cdr sequence)))))
```

```
(sum (map square (prime-numbers 7 0)))
```

Do wyjaśnienia:

- ~~co znaczy „suma *X*”?~~
- ~~co znaczy „kwadraty *Y*”~~
- ~~co znaczy „siedem początkowych liczb pierwszych”?~~

Jak inaczej można pomyśleć „siedem początkowych liczb pierwszych”?

```
#lang lazy
(define (numbers-from n)
  (cons n (numbers-from (+ n 1))))
(define natural-numbers (numbers-from 0))
(define primes
  (filter prime? natural-numbers))
(sum (map square (take 7 primes)))
```

Jak inaczej można pomyśleć „siedem początkowych liczb pierwszych”?

```
#lang lazy
(define (numbers-from n)
  (cons n (numbers-from (+ n 1))))
(define natural-numbers (numbers-from 0))
(define primes
  (filter prime? natural-numbers))
(sum (map square (take 7 primes)))
```

Jak inaczej można pomyśleć „siedem początkowych liczb pierwszych”?

```
#lang lazy
(define (numbers-from n)
  (cons n (numbers-from (+ n 1))))
(define natural-numbers (numbers-from 0))
(define primes
  (filter prime? natural-numbers))
(sum (map square (take 7 primes)))
```

Jak inaczej można pomyśleć „siedem początkowych liczb pierwszych”?

```
#lang lazy
(define (numbers-from n)
  (cons n (numbers-from (+ n 1))))
(define natural-numbers (numbers-from 0))
(define primes
  (filter prime? natural-numbers))
(sum (map square (take 7 primes)))
```

Jak inaczej można pomyśleć „siedem początkowych liczb pierwszych”?

```
#lang lazy
(define (numbers-from n)
  (cons n (numbers-from (+ n 1))))
(define natural-numbers (numbers-from 0))
(define primes
  (filter prime? natural-numbers))
(sum (map square (take 7 primes)))
```

Jak inaczej można pomyśleć „siedem początkowych liczb pierwszych”?

```
#lang lazy
(define (numbers-from n)
  (cons n (numbers-from (+ n 1))))
(define natural-numbers (numbers-from 0))
(define primes
  (filter prime? natural-numbers))
(sum (map square (take 7 primes)))
```

Jak inaczej można pomyśleć „siedem początkowych liczb pierwszych”?

```
#lang lazy
(define (numbers-from n)
  (cons n (numbers-from (+ n 1))))
(define natural-numbers (numbers-from 0))
(define primes
  (filter prime? natural-numbers))
(sum (map square (take 7 primes)))
```

```
(define (filter condition sequence)
  (if (equal? sequence '())
    '()
    (if (condition (car sequence))
      (cons (car sequence)
          (filter condition (cdr sequence)))
      (filter condition (cdr sequence)))))
(define (take n l)
  (if (equal? n 0)
    '()
    (cons (car l)
        (take (- n 1) (cdr l)))))
```

```
(define (filter condition sequence)
  (if (equal? sequence '())
    '()
    (if (condition (car sequence))
      (cons (car sequence)
          (filter condition (cdr sequence)))
      (filter condition (cdr sequence)))))
(define (take n l)
  (if (equal? n 0)
    '()
    (cons (car l)
        (take (- n 1) (cdr l)))))
```

```
(define (filter condition sequence)
  (if (equal? sequence '())
    '()
    (if (condition (car sequence))
      (cons (car sequence)
          (filter condition (cdr sequence)))
      (filter condition (cdr sequence)))))
(define (take n l)
  (if (equal? n 0)
    '()
    (cons (car l)
        (take (- n 1) (cdr l)))))
```

Czy nie łatwiej pozostać przy programowaniu imperatywnym?

- za dużo sposobów przekazywania informacji pomiędzy modułami (zmienne globalne, argumenty wynikowe, parametry konfiguracyjne) – duża entropia kodu
- konieczność specyfikowania szczegółów, które nie są istotne dla problemu (przepływ sterowania, kolejność wykonywania obliczeń)

Czy nie łatwiej pozostać przy programowaniu imperatywnym?

- za dużo sposobów przekazywania informacji pomiędzy modułami (zmienne globalne, argumenty wynikowe, parametry konfiguracyjne) – duża entropia kodu
- konieczność specyfikowania szczegółów, które nie są istotne dla problemu (przepływ sterowania, kolejność wykonywania obliczeń)

Czy nie łatwiej pozostać przy programowaniu imperatywnym?

- za dużo sposobów przekazywania informacji pomiędzy modułami (zmienne globalne, argumenty wynikowe, parametry konfiguracyjne) – duża entropia kodu
- konieczność specyfikowania szczegółów, które nie są istotne dla problemu (przepływ sterowania, kolejność wykonywania obliczeń)

Lisp

Czy te dziwne nawiasy są konieczne?

Odnoszenie się do wyrażeń językowych, np.

> *„znaczenie wyrażenia «0 najmniejszych liczb pierwszych większych od M» jest tożsame ze znaczeniem wyrażenia «pusty ciąg»”*

> *„zdanie «śnieg jest biały» jest prawdziwe wtedy i tylko wtedy, gdy śnieg jest biały”*

Lisp

Czy te dziwne nawiasy są konieczne?
Odnoszenie się do wyrażeń językowych, np.

> *„znaczenie wyrażenia «0 najmniejszych liczb pierwszych większych od M» jest tożsame ze znaczeniem wyrażenia «pusty ciąg»”*

> *„zdanie «śnieg jest biały» jest prawdziwe wtedy i tylko wtedy, gdy śnieg jest biały”*

Lisp

Czy te dziwne nawiasy są konieczne?
Odnoszenie się do wyrażeń językowych, np.

> *„znaczenie wyrażenia «0 najmniejszych liczb pierwszych większych od M» jest tożsame ze znaczeniem wyrażenia «pusty ciąg»”*
>
> *„zdanie «śnieg jest biały» jest prawdziwe wtedy i tylko wtedy, gdy śnieg jest biały”*

Lisp

Czy te dziwne nawiasy są konieczne?
Odnoszenie się do wyrażeń językowych, np.

> *„znaczenie wyrażenia «0 najmniejszych liczb pierwszych większych od M» jest tożsame ze znaczeniem wyrażenia «pusty ciąg»”*
>
> *„zdanie «śnieg jest biały» jest prawdziwe wtedy i tylko wtedy, gdy śnieg jest biały”*

Operator `quote`

Operator `quote` w Lispie zapobiega ewaluacji wyrażenia.

Na przykład wartością wyrażenia

```
(quote (sum (map square (prime-numbers 7 0))))
```

jest sekwencja dwuelementowa, której pierwszym elementem jest symbol `sum`, a drugim sekwencja trójelementowa, której pierwszym elementem...

`(quote x)` skracamy jako `'x`

Zauważmy, że:

1. programy, z którymi mieliśmy do tej pory styczność, opierały się na przetwarzaniu sekwencji
2. teksty programów, które pisaliśmy, same są (pozagnieżdżanymi) sekwencjami (homoikoniczność)

Operator `quote`

Operator `quote` w Lispie zapobiega ewaluacji wyrażenia.
Na przykład wartością wyrażenia
`(quote (sum (map square (prime-numbers 7 0))))`
jest sekwencja dwuelementowa, której pierwszym elementem jest symbol `sum`, a drugim sekwencja trójelementowa, której pierwszym elementem...
`(quote x)` skracamy jako `'x`
Zauważmy, że:

1. programy, z którymi mieliśmy do tej pory styczność, opierały się na przetwarzaniu sekwencji
2. teksty programów, które pisaliśmy, same są (pozagnieżdżanymi) sekwencjami (homoikoniczność)

Operator `quote`

Operator `quote` w Lispie zapobiega ewaluacji wyrażenia.
Na przykład wartością wyrażenia
`(quote (sum (map square (prime-numbers 7 0))))`
jest sekwencja dwuelementowa, której pierwszym elementem jest symbol `sum`, a drugim sekwencja trójelementowa, której pierwszym elementem...
`(quote x)` skracamy jako `'x`
Zauważmy, że:

1. programy, z którymi mieliśmy do tej pory styczność, opierały się na przetwarzaniu sekwencji
2. teksty programów, które pisaliśmy, same są (pozagnieżdżanymi) sekwencjami (homoikoniczność)

Operator `quote`

Operator `quote` w Lispie zapobiega ewaluacji wyrażenia.
Na przykład wartością wyrażenia
`(quote (sum (map square (prime-numbers 7 0))))`
jest sekwencja dwuelementowa, której pierwszym elementem jest symbol `sum`, a drugim sekwencja trójelementowa, której pierwszym elementem...
`(quote x)` skracamy jako `'x`
Zauważmy, że:

1. programy, z którymi mieliśmy do tej pory styczność, opierały się na przetwarzaniu sekwencji
2. teksty programów, które pisaliśmy, same są (pozagnieżdżanymi) sekwencjami (homoikoniczność)

Operator `quote`

Operator `quote` w Lispie zapobiega ewaluacji wyrażenia.
Na przykład wartością wyrażenia
`(quote (sum (map square (prime-numbers 7 0))))`
jest sekwencja dwuelementowa, której pierwszym elementem jest symbol `sum`, a drugim sekwencja trójelementowa, której pierwszym elementem...
`(quote x)` skracamy jako `'x`
Zauważmy, że:

1. programy, z którymi mieliśmy do tej pory styczność, opierały się na przetwarzaniu sekwencji
2. teksty programów, które pisaliśmy, same są (pozagnieżdżanymi) sekwencjami (homoikoniczność)

Operator `quote`

Operator `quote` w Lispie zapobiega ewaluacji wyrażenia.
Na przykład wartością wyrażenia
`(quote (sum (map square (prime-numbers 7 0))))`
jest sekwencja dwuelementowa, której pierwszym elementem jest symbol `sum`, a drugim sekwencja trójelementowa, której pierwszym elementem...
`(quote x)` skracamy jako `'x`
Zauważmy, że:

1. programy, z którymi mieliśmy do tej pory styczność, opierały się na przetwarzaniu sekwencji
2. teksty programów, które pisaliśmy, same są (pozagnieżdżanymi) sekwencjami (homoikoniczność)

Operator `quote`

Operator `quote` w Lispie zapobiega ewaluacji wyrażenia.
Na przykład wartością wyrażenia
`(quote (sum (map square (prime-numbers 7 0))))`
jest sekwencja dwuelementowa, której pierwszym elementem jest symbol `sum`, a drugim sekwencja trójelementowa, której pierwszym elementem...
`(quote x)` skracamy jako `'x`
Zauważmy, że:

1. programy, z którymi mieliśmy do tej pory styczność, opierały się na przetwarzaniu sekwencji
2. teksty programów, które pisaliśmy, same są (pozagnieżdżanymi) sekwencjami (homoikoniczność)

Homoikoniczność

Co nam to daje?

- składnia uniwersalna
- opisywanie języków
- pisanie *programów, które piszą programy*
- rozszerzanie składni języka
- lepsze narzędzia
- możliwość wykomentowywania pojedynczych wyrażeń

Homoikoniczność

Co nam to daje?

- składnia uniwersalna
- opisywanie języków
- pisanie *programów, które piszą programy*
- rozszerzanie składni języka
- lepsze narzędzia
- możliwość wykomentowywania pojedynczych wyrażeń

Homoikoniczność

Co nam to daje?

- składnia uniwersalna
- opisywanie języków
- pisanie *programów, które piszą programy*
- rozszerzanie składni języka
- lepsze narzędzia
- możliwość wykomentowywania pojedynczych wyrażeń

Homoikoniczność

Co nam to daje?

- składnia uniwersalna
- opisywanie języków
- pisanie *programów, które piszą programy*
- rozszerzanie składni języka
- lepsze narzędzia
- możliwość wykomentowywania pojedynczych wyrażeń

Homoikoniczność

Co nam to daje?

- składnia uniwersalna
- opisywanie języków
- pisanie *programów, które piszą programy*
- rozszerzanie składni języka
- lepsze narzędzia
- możliwość wykomentowywania pojedynczych wyrażeń

Homoikoniczność

Co nam to daje?

- składnia uniwersalna
- opisywanie języków
- pisanie *programów, które piszą programy*
- rozszerzanie składni języka
- lepsze narzędzia
- możliwość wykomentowywania pojedynczych wyrażeń

Homoikoniczność

Co nam to daje?

- składnia uniwersalna
- opisywanie języków
- pisanie *programów, które piszą programy*
- rozszerzanie składni języka
- lepsze narzędzia
- możliwość wykomentowywania pojedynczych wyrażeń

Co dalej?

- **Struktura i Interpretacja Programów Komputerowych**
  `https://mitpress.mit.edu/sicp`
- **A Pamphlet against R**
  `http://panicz.github.io/pamphlet/`
  - algorytmy genetyczne
  - logika rozmyta
  - drzewa decyzyjne
  - klasteryzacja
- warsztaty?

Co dalej?

- **Struktura i Interpretacja Programów Komputerowych**
  `https://mitpress.mit.edu/sicp`
- **A Pamphlet against R**
  `http://panicz.github.io/pamphlet/`
  - algorytmy genetyczne
  - logika rozmyta
  - drzewa decyzyjne
  - klasteryzacja
- warsztaty?

Co dalej?

- **Struktura i Interpretacja Programów Komputerowych**
  `https://mitpress.mit.edu/sicp`
- **A Pamphlet against R**
  `http://panicz.github.io/pamphlet/`
  - algorytmy genetyczne
  - logika rozmyta
  - drzewa decyzyjne
  - klasteryzacja
- warsztaty?

Co dalej?

- **Struktura i Interpretacja Programów Komputerowych**
  `https://mitpress.mit.edu/sicp`
- **A Pamphlet against R**
  `http://panicz.github.io/pamphlet/`
  - algorytmy genetyczne
  - logika rozmyta
  - drzewa decyzyjne
  - klasteryzacja
- warsztaty?

Co dalej?

- **Struktura i Interpretacja Programów Komputerowych**
  `https://mitpress.mit.edu/sicp`
- **A Pamphlet against R**
  `http://panicz.github.io/pamphlet/`
  - algorytmy genetyczne
  - logika rozmyta
  - drzewa decyzyjne
  - klasteryzacja
- warsztaty?

Co dalej?

- **Struktura i Interpretacja Programów Komputerowych**
  `https://mitpress.mit.edu/sicp`
- **A Pamphlet against R**
  `http://panicz.github.io/pamphlet/`
  - algorytmy genetyczne
  - logika rozmyta
  - drzewa decyzyjne
  - klasteryzacja
- warsztaty?

Co dalej?

- **Struktura i Interpretacja Programów Komputerowych**
  `https://mitpress.mit.edu/sicp`
- **A Pamphlet against R**
  `http://panicz.github.io/pamphlet/`
  - algorytmy genetyczne
  - logika rozmyta
  - drzewa decyzyjne
  - klasteryzacja
- warsztaty?

Co dalej?

- **Struktura i Interpretacja Programów Komputerowych**
  `https://mitpress.mit.edu/sicp`
- **A Pamphlet against R**
  `http://panicz.github.io/pamphlet/`
  - algorytmy genetyczne
  - logika rozmyta
  - drzewa decyzyjne
  - klasteryzacja
- warsztaty?

Koniec?

Dziękuję za uwagę.