Podstawy programowania w języku Nim

podręcznik dla początkujących

Namysław Tatarynowicz

Zaktualizowano: 14 lutego 2024

main.nim

Spis treści

1 Wprowadzenie do języka Nim

1.1 Wstęp

Uwaga

Ten podręcznik nie jest oficjalnym dokumentem i nie jest powiązany z `nim-lang.org`.

Tworząc ten dokument, przyjąłem sobie za cel przedstawienie podstaw programowania w Nim. Jest to również sposób na moją naukę tego języka. Zawartość dokumentu będzie ewoluowała wraz ze wzrostem mojej świadomości programowania w Nim.

Niniejszy podręcznik jest we wczesnej fazie tworzenia. Zaplanowałem przygotowanie funkcjonalnego materiału do dalszej rozbudowy (w tym ciągłą poprawę istniejących już punktów). Gdybyś jednak Drogi Czytelniku zauważył merytoryczny błąd lub miałbyś jakąś sugestię, proszę o kontakt na adres mailowy: `n.tatarynowicz@gmail.com`.

Etapy tworzenia podręcznika przedstawiają się następująco:

- ☑ Określenie ogólnej koncepcji tematycznej
- ☑ Wybór techniki realizacji
- ☑ Określenie ostatecznej formy graficznej
- ☐ Opracowanie treści
- ☐ Opracowanie okładki
- ☐ Publikacja w sieci

Opracowując ten dokument, korzystam z następującego oprogramowania:

```
system operacyjny: antiX 22
język            : Nim 2.0.2
kompilator       : glibc 2.36
edytor kodu      : Geany 1.38
```

1.2 Poznajmy Nima

Nim jest językiem programowania o składni zbliżonej do języka Python, ale kompilowanym. Dzięki temu umożliwia tworzenie programów o podobnej wydajności jak programy napisane w języku C. Nim do kompilacji kodu wykorzystuje jedną z dostępnych bibliotek języka C/C++ (domyślnie C). Zatem, podczas kompilacji programu, najpierw powstaje plik C i dopiero ten plik jest kompilowany. Ostatecznie Nim daje możliwość kompilacji do C, C++ lub JavaScript.

Podobnie, jak w języku Python, w Nim także możemy korzystać z gotowych modułów, które importujemy tak samo w obu językach. Trzeba zauważyć, że Nim nie posiada tak dużej ich liczby, jak Python.

Najnowszą wersję Nima znajdziemy na stronie `https://nim-lang.org/`. W momencie tworzenia tego dokumentu, na stronie twórców, Nim jest dostępny w wersji 2.0.2.

Znajduje się tam również środowisko, dzięki któremu będziemy mogli skompilować i uruchomić nasz kod on-line oraz wygenerować odnośnik i go udostępnić: `https://play.nim-lang.org/`.

Język Nim jest wolnym oprogramowaniem — jest rozprowadzany na licencji MIT.

Jeśli napotkamy na jakiś programistyczny problem, to dzięki społeczności Nima możemy uzyskać pomoc w jego rozwiązaniu. Poniżej umieszczam listę miejsc, które możemy odwiedzić. W momencie przygotowywania tego podręcznika nie spotkałem polskojęzycznych kanałów społecznościowych przeznaczonych użytkownikom Nima.

blog:
`https://nim-lang.org/blog.html`

forum:
`https://forum.nim-lang.org/`

IRC:
`irc.libera.chat, #nim`

Discord:
`https://discord.gg/nim`

Matrix:
`https://matrix.to/#/#nim-lang:matrix.org`

Telegram:
`https://t.me/nim_lang`

Twitter:
`https://twitter.com/nim_lang`

Reddit:
`https://www.reddit.com/r/nim`

StackOverflow:
`https://stackoverflow.com/questions/tagged/nim-lang`

Wiki:
`https://github.com/nim-lang/Nim/wiki`

1.3 Przygotowanie środowiska

Abyśmy mogli zacząć pisać i kompilować programy na naszym komputerze, będziemy potrzebowali:

- pakietu Nim
- kompilatora języka C/C++
- dowolnego/ulubionego edytora kodu

Nim nie dostarcza zintegrowanego środowiska programistycznego, dlatego wszystkie powyższe elementy będziemy musieli pobrać, zainstalować i skonfigurować samodzielnie.

W przypadku systemu operacyjnego Windows ze strony projektu należy pobrać archiwum zip Nima w wersji odpowiedniej do posiadanej architektury komputera. Analogicznie należy postąpić ze znajdującym się na stronie archiwum kompilatora języka C (mingw).

Chcąc zainstalować Nima na jednej z dystrybucji GNU/Linux można to zrobić na dwa sposoby. Możemy zainstalować pakiet Nim z repozytorium danej dystrybucji lub pobrać go ze strony projektu. W obu przypadkach dodatkowo będzie potrzebny kompilator C/C++. Warto wówczas zainstalować metapakiet o nazwie `build-essential` zawierający niezbędne biblioteki wykorzystywane do pracy z językami C/C++.

1.4 Kompilacja

Najszybszym sposobem skompilowania przygotowanego pliku jest wykonanie polecenia:

```
nim c nazwa_pliku.nim
```

Po poprawnym skompilowaniu otrzymamy wykonywalny plik o takiej samej nazwie.

Ogólna składnia kompilacji programu jest następująca:
`nim polecenie [opcje] [nazwa_pliku] [argumenty]`

`polecenie` może przyjmować następujące wartości:

`compile, c`	kompiluje projekt z wykorzystaniem domyślnego generatora kodu C
`r`	kompiluje do $nimcache/projname i uruchamia z argumentami domyślnie używając kodu C
`doc`	generuje dokumentację dla pliku wejściowego

`opcje` mogą przyjmować następujące wartości:

`-p, --path:PATH`	dodaj ścieżkę do ścieżek wyszukiwania
`-d, --define:SYMBOL(:VAL)`	zdefiniuj symbol warunkowy
`-u, --undef:SYMBOL`	usuń definicję symbolu warunkowego
`-f, --forceBuild:on|off`	wymuś przebudowę wszystkich modułów
`--stackTrace:on|off`	włącz/wyłącz ślad stosu
`--threads:on|off`	włącz/wyłącz obsługę wielowątkowości
`-x, --checks:on|off`	włącz/wyłącz wszystkie kontrole środowiska uruchomieniowego
`-a, --assertions:on|off`	włącz/wyłącz asercje
`--opt:none|speed|size`	nie optymalizuj, optymalizuj pod względem szybkości lub czasu
`--debugger:native`	użyj natywnego debugera (gdb)
`--app:console|gui|lib|staticlib`	wygeneruj aplikację konsolową, graficzną, DLL lub bibliotekę statyczną
`-r, --run`	uruchom skompilowany program z podanymi argumentami
`--eval:cmd`	bezpośrednia ewaluacja kodu nim; np.: `nim --eval:"echo 1"`
`--fullhelp`	pokaż pełny podręcznik pomocy
`-h, --help`	pokaż podręcznik pomocy
`-v, --version`	pokaż szczegóły dot. zainstalowanej wersji

Przykładowa konstrukcja polecenia do skompilowania kodu zawartego w pliku `main.nim`:

```
nim c -r --opt:speed main.nim
```

Dzięki czemu kod (zapisany w pliku `main.nim`) najpierw zostanie skompilowany (`c`) z optymalizacją pod względem szybkości działania (`--opt:speed`), a następnie uruchomiony (`-r`).

1.5 Pierwszy program

Zazwyczaj podczas nauki programowania nowego języka, jako pierwszy, pisze się program wyświetlający napis `Hello world`. Więc tradycji niech stanie się zadość.

Ważne

Wszystkie pliki źródłowe Nima muszą być kodowane z wykorzystaniem UTF-8 (lub jego podzbiorze ASCII). Żadne inne kodowanie nie jest wspierane.

Do wyświetlania tekstu na ekranie służy polecenie `echo`. Konsekwencją jego działania, poza wyświetlaniem tekstu, jest przejście kursora do nowej linii.

```
echo "Hello world"
```

```
Hello world
```

Istnieją różne sposoby wyświetlania tekstu w konsoli:

```
echo "Hello world"
echo("Hello world")
"Hello world".echo()
"Hello world".echo
"Hello ".echo("world")
"Hello".echo " world"
```

Wszystkie powyższe przykłady wyświetlą napis `Hello world`, po czym kursor przejdzie do nowej linii.

Jeśli chcemy, aby po wyświetleniu tekstu kursor pozostał w tej samej linii, skorzystamy z `stdout.write`. Możemy i tu wymusić przejście do nowej linii używając `\n`

```
stdout.write "Hello world. "
stdout.write "Bye!\nBye!"
```

```
Hello world. Bye!
Bye!
```

Można oczywiście wyświetlać liczby lub wyniki operacji arytmetycznych. Liczby zmiennoprzecinkowe mają oddzieloną część całkowitą od dziesiętnej kropką.

```
echo 20
echo 3.14
echo 5+5
echo 6-2
echo 2*4
echo 15/2
```

```
20
3.14
10
4
8
7.5
```

Pisząc programy, możemy używać następujących operatorów arytmetycznych:

Operator	Nazwa
`+`	dodawanie
`-`	odejmowanie
`*`	mnożenie
`/`	dzielenie
`div`	dzielenie bez reszty
`mod`	reszta z dzielenia

W kodzie źródłowym można stosować komentarze. W Nim mamy do dyspozycji ich dwa rodzaje — jednolinijkowe, zaczynające się symbolem hash `#` i wielolinijkowe — tekst zapisany pomiędzy `#[` a `]#` lub pomiędzy `discard """` a `"""`. Poniższy przykład przedstawia wyżej opisane sposoby.

```
# To jest komentarz jednolinijkowy

echo "Hej!" # można też tak komentować

#[
To jest
komentarz
wielolinijkowy
]#

discard """
To także jest
komentarz
wielolinijkowy
"""
```

Po kompilacji i uruchomieniu pliku zostanie wyświetlony tylko napis "`Hej!`".

Edytory kodu zwykle posiadają własną opcję wstawiania komentarzy, np. generują znaki komentowania jednolinijkowego, ale dla zaznaczonej części kodu. Taki sposób wydaje się najbardziej wygodny i szybki.

Podobnie, jak w innych językach programowania, także i w Nim można łączyć teksty (ciągi znaków). Takie łączenie nazywa się konkatenacją.

Gdy mamy przygotowane już napisy do połączenia, wówczas możemy użyć znaku `&`.

```
echo "Ala " & "ma " & "kota."
```

```
Ala ma kota.
```

Bezpośrednio można łączyć ze sobą tylko ciągi znaków. Jeśli mamy do złączenia ciąg znaków i liczbę, wówczas trzeba tę liczbę zrzutować na ciąg znaków. Do rzutowania danego typu danych na ciąg znaków użyjemy symbolu `$`, który wstawimy przed liczbą.

```
echo "Ala " & "ma " & $20 & " lat."
```

```
Ala ma 20 lat.
```

Można też to zrobić trochę szybciej, używając znaku `,`, który automatycznie zamienia wszystkie argumenty na ciągi znaków.

```
echo "Ala ", "ma ", 20, " lat."
```

```
Ala ma 20 lat.
```

Na tym etapie nauki mogę zasugerować przyjęcie jednego, wygodnego sposobu zarówno wyświetlania tekstów, tworzenia komentarzy, zwłaszcza wielolinijkowych, jak i łączenia tekstów. Pomoże to w utrzymaniu porządku w naszym kodzie i sprawi, że będzie on bardziej czytelny.

2 Zmienne, stałe i typy danych

2.1 Zmienne i stałe

Zmienne w programowaniu często porównuje się do pojemników, które przechowują różne wartości. Tymi wartościami może być pojedynczy znak, ciąg znaków, różnego rodzaju liczby, a nawet wartości logiczne — prawda, czy fałsz. Odwołując się do tych pojemników, możemy użyć wartości w nich przechowywanych.

Przed użyciem zmiennej trzeba ją wcześniej zadeklarować. W Nim mamy do dyspozycji dwa rodzaje zmiennych — mutowalne i niemutowalne.

Zmienne mutowalne, to takie, do których możemy wielokrotnie przypisywać wartości.

Z kolei zmienne niemutowalne, to takie, do których jeśli raz przypiszemy wartość, nie będziemy mogli jej zmienić.

Do deklaracji zmiennej mutowalnej służy `var`. Spójrzmy na przykład:

```
var name = "Adam"

echo "Cześć! Mam na imię ", name, "."
```

```
Cześć! Mam na imię Adam.
```

Deklaracja zmiennej polega na podaniu nazwy, pod jaką będzie ona przechowywała wartość oraz typu tych danych. Nim „potrafi" rozpoznać, jakiego typu jest zmienna, którą przechowujemy. Jednak, gdy chcemy zadeklarować zmienną bez podawania jej wartości, musimy sami określić jej typ:

```
var age: int
```

Właśnie zadeklarowaliśmy zmienną o nazwie `age`, która będzie przechowywała liczby całkowite (`int`).

Nic nie stoi na przeszkodzie, żeby oprócz określenia typu zmiennej przypisać od razu do niej wartość:

```
var age: int = 20
```

Po deklaracji zmiennej można zmieniać jej wartość, ale pamiętajmy, że nowe wartości muszą być tego samego typu.

```
var age: int = 20
echo age
age = 17
echo age
```

```
20
17
```

Jeśli deklarujemy kilka zmiennych, możemy uprościć zapis, stosując blok utworzony z wcięć:

```
var
  x = 10
  y = 3.5
  z = "Hello!"
```

Możemy również używać niemutowalnych (niemodyfikowalnych) zmiennych. Deklarujemy je, używając `let`:

```
let sum = 4
```

W Nim możemy tworzyć zmienne lub stałe, których nazwy składają się z kilku słów, pomiędzy którymi może być wstawiona spacja. Musimy wówczas nazwę zmiennej zamknąć pomiędzy ` a `. Co więcej, odwołując się do nazw zmiennych lub stałych Nim nie uwzględnia wielkości znaków poza pierwszym znakiem. Proszę zwrócić uwagę na poniższy przykład:

```
let `my own variable` = 12

echo `my own variable`
echo myownvariable
echo myOwnVariable
```

```
12
12
12
```

Gdybyśmy jednak próbowali odwołać się do zmiennej o nazwie `MyOwnVariable` otrzymalibyśmy błąd: `undeclared identifier: 'MyOwnVariable'`.

Oprócz zmiennych w Nim możemy używać stałych. Służą one do przechowywania wartości, których, gdy zostaną już określone, nie będzie możliwości dokonania ich zmiany.

```
const pi = 3.14159
```

Proszę zwrócić uwagę, że **każda stała zadeklarowana w ten sposób musi mieć ustaloną wartość już podczas kompilacji programu.**

Podobnie, jak w przypadku zmiennych, gdy mamy więcej niż jedną stałą do utworzenia, możemy tego dokonać w bloku kodu:

```
const
  pi = 3.14159
  c = 300_000_000
  text = "Hello world"
```

W powyższym przykładzie został użyty trochę inny, niż do tej pory sposób zapisu liczb: `300_000_000`. Ułatwia on poprawne wpisywanie bardzo dużych (lub bardzo małych) wartości, ale nie zmienia on interpretacji tej liczby. Przykład:

```
echo 300_000_000
```

```
300000000
```

2.2 Typy danych

W Nim możemy korzystać z następujących typów danych:

```
var c: char = 'y'          # pojedynczy znak
var name: string = "Adam" # ciąg znaków
var age: int = 10          # liczba całkowita
var pi: float = 3.14159    # liczba zmiennoprzecinkowa
var b: bool = true         # wartość logiczna
```

Podczas deklaracji zmiennej lub stałej przechowującej pojedynczy znak zamiast cudzysłowów używamy apostrofów.

Żeby sprawdzić typ danej zmiennej lub stałej korzystamy z atrybutu `type`. Oto typy danych dla zmiennych z powyższego przykładu:

```
echo c.type
echo name.type
echo age.type
echo pi.type
echo b.type
```

```
char
string
int
float
bool
```

W przypadku liczb całkowitych możemy dodatkowo określić zakres. Mamy do dyspozycji: `int64`, `int32`, `int16`, `int8`. Zatem możemy ustalić zakresy liczb całkowitych, kolejno: 64 bitowe, 32 bitowe, 16 bitowe i 8 bitowe.

Analogicznie mamy do dyspozycji liczby całkowite bez znaku (liczby dodatnie i 0): `uint64`, `uint32`, `uint16`, `uint8`.

Możemy sprawdzić, jak dużo miejsca w pamięci zajmuje stała lub zmienne. Wykorzystamy do tego operator `sizeof`. Wartość ta jest podawana w bajtach.

```
var a: uint64 = 10
var b: uint32 = 10
var c: uint16 = 10
var d: uint8 = 10

echo a.sizeof
echo b.sizeof
echo c.sizeof
echo d.sizeof
```

```
8
4
2
1
```

Dobranie odpowiedniego typu do przechowywania danych pozwala zoptymalizować ilość pamięci potrzebną do działania naszego programu.

3 Wprowadzanie danych przez użytkownika

W tym krótkim punkcie przedstawię, w jaki sposób można przekazywać dane w konsoli.

Na początek napiszemy program, który poprosi użytkownika o podanie swojego imienia. Na potrzeby tego przykładu niech tym imieniem będzie Adam. A następnie przywita się z Adamem.

```
stdout.write("Podaj swoje imię: ")
let name = stdin.readLine()

echo name.type
echo "Witaj, ", name, "!"
```

```
Podaj swoje imię: Adam
string
Witaj, Adam!
```

Wszystkie dane przekazywane w ten sposób w konsoli są pobierane, jako `string` (ciąg znaków). Dlatego, gdybyśmy chcieli użyć ich do obliczeń, trzeba je najpierw przekonwertować do odpowiedniego typu.

Dla przykładu pobierzmy liczbę od użytkownika i wykonajmy na niej działanie — pomnóżmy ją przez dwa. Przy okazji pierwszy raz zaimportujemy moduł, będzie to `std/strutils`, do naszego programu (o modułach powiem w sekcji Moduły), potrzebny do wykorzystania funkcji `parseInt()` konwertującej tekst na liczbę całkowitą. Prześledźmy poniższy kod:

```
import strutils

stdout.write("Podaj liczbę: ")
let number = readLine(stdin).parseInt()

echo number*2
```

```
4
```

W powyższym przykładzie została podana liczba 2, więc po pomnożeniu przez 2 otrzymaliśmy 4.

Można także oczekiwać na pobranie pojedynczych znaków zamiast całego ciągu znaków. Skorzystamy wówczas z funkcji `stdin.readChar()`:

```
stdout.write "Napisz literę i naciśnij enter: "
var letter = stdin.readChar()

echo letter
```

```
Napisz literę i naciśnij enter: a
a
```

4 Praca z tekstami

W języku Nim możemy odwoływać się do poszczególnych znaków tekstu lub do pozycji (tzw. numer indeksu), na których one się znajdują. Warto zapamiętać, że **numeracja indeksów rozpoczyna się od 0**. Poniżej umieszczam przykładowe zestawienie.

```
let text = "Hello world"

echo text[0]          # pierwszy znak
echo text[1..8]       # znaki od indeksu 1 do 8
echo text[^1]         # ostatni znak
echo text[^2]         # przedostatni znak
echo text[^3..^1]     # ostatnie trzy znaki
echo text.low         # pierwszy indeks
echo text.high        # ostatni indeks
echo text[text.low]   # znak spod pierwszego indeksu
echo text[text.high] # znak spod ostatniego indeksu
```

```
H
ello wor
d
l
rld
0
10
H
d
```

Używając funkcji `add()`, można w łatwy sposób dołączać nowe ciągi znaków do już istniejących tzw. metodą inline, czyli bezpośrednio nadpisywać wartości zmiennej.

```
var text = "Lorem"
text.add(" ipsum")

echo text
```

```
Lorem ipsum
```

Do pomiaru długości ciągu znaków służy funkcja `len()`. Można z niej skorzystać na różne sposoby.

```
var text = "Lorem"

echo len text
echo len(text)
echo text.len
echo text.len()
```

Wszystkie powyższe przykłady wyświetlą długość tekstu zapisanego pod zmienną `text`, czyli 5.

Możemy zmienić długość wartości zmiennej tekstowej. W tym celu użyjemy `setlen()`. Zmieńmy długość ciągu na 3 znaki.

```
var text = "Lorem ipsum"
text.setlen(3)

echo text
```

```
Lor
```

Istnieje również możliwość tworzenia wielolinijkowych ciągów znaków, które zachowują także znaki nowej linii. Należy wówczas użyć potrójnych cudzysłowów — zamknąć tekst pomiędzy `"""` a `"""`.

```
let text = """
Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt...
"""

echo text
```

```
Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt...
```

Wyświetlając teksty w konsoli, możemy używać znaków specjalnych, tych poprzedzonych symbolem backslash `\`. Takim znakiem specjalnym jest znany już nam z początku podręcznika znak nowej linii `\n`. Innym znakiem specjalnym jest `\t` wstawiający znak tabulacji. Poniższy przykład pokazuje, jak w konsoli będą wyglądały słowa oddzielone takim właśnie znakiem.

```
echo "pon.\two.\tśr.\tczw.\tpią."
```

```
pon.    wto.    śr.    czw.    pią.
```

Symbol backslash oprócz tego, że czyni ze zwykłych znaków znaki specjalne, może także zamieniać znaki specjalne na zwykłe. Np. gdybyśmy chcieli wyświetlić tekst, który zawiera w sobie cudzysłów, wówczas musimy poprzedzić ten cudzysłów znakiem backslash. Oto przykład:

```
echo "Oto cytat: \"Carpe diem\"."
```

```
Oto cytat: "Carpe diem".
```

Wcześniej poznaliśmy jak łączyć ze sobą ciągi znaków oraz ciągi z liczbami. Jednak gdy mamy do złączenia więcej elementów, np. tekst i zmienne, wygodniej jest użyć formatowania z modułu `strformat`.

```
import std/strformat

let
  name = "Adam"
  age = 19
  hobby = "programowanie w Nim"

echo &"{name} ma {age} lat i jego hobby to {hobby}."
```

```
Adam ma 19 lat i jego hobby to programowanie w Nim.
```

Zamiast znaku `&` w powyższym przykładzie, można użyć `fmt`, co przyniesie taki sam efekt. Różnica w tych dwóch sposobach formatowania tekstów jest taka, że `fmt` nie interpretuje znaków specjalnych i wyświetla je, jak surowy tekst. Oto przykład:

```
import std/strformat

echo &"Ala ma\tkota."
echo fmt"Ala ma\tkota."
```

```
Ala ma    kota.
Ala ma\tkota.
```

Dobrym zwyczajem jest, żeby liczba wpisanych znaków w jednej linijce kodu nie przekraczała 80. Pomaga to w utrzymaniu przejrzystości w kodzie. Czasem jednak zdarzy się, że będziemy musieli użyć dłuższego tekstu, przekraczającego 80 znaków. Wygodnie będzie go podzielić na mniejsze części, które jednak będą stanowiły całość przy wyświetlaniu go w terminalu:

```
let long_str = "Za siedmioma górami, za siedmioma lasami,"&
               " za siedmioma morzami,"&
               " żyła sobie mała dziewczynka."
```

```
Za siedmioma górami, za siedmioma lasami, za siedmioma
morzami, żyła sobie mała dziewczynka.
```

Gdyby długi ciąg znaków miałby powstać z różnych typów danych, moglibyśmy użyć formatowania i wykorzystać moduł `std/strformat`. Wówczas nasz kod będzie wyglądał tak:

```
import std/strformat

let
  name = "Adam"
  age = 17
  hobby = "programowanie w Nim"
  long_str = &"{name} mieszka w Polsce i ma {age} lat."&
             &" Bardzo lubi grać w piłkę."&
             &" Ostatnio interesuje go {hobby}."

echo long_str
```

```
Adam mieszka w Polsce i ma 17 lat. Bardzo lubi grać w piłkę.
Ostatnio interesuje go programowanie w Nim.
```

5 Instrukcje warunkowe

Instrukcje warunkowe pozwalają komputerowi na podejmowanie decyzji. Prześledźmy poniższy pseudokod:

```
if warunek:
  jeśli warunek jest spełniony, realizowany jest ten blok
elif warunek:
  jeśli powyższy warunek nie jest spełniony,
  sprawdzany jest ten warunek i jeśli jest spełniony
  realizowany jest ten blok
else:
  jeśli żaden z powyższych warunków nie jest spełniony,
  realizowany jest ten blok
```

W powyższym przykładzie, po dwukropkach, zostały użyte tzw. wcięcia. W Nim, podobnie jak w Pythonie, wcięcia służą do wydzielania części kodu, który ma zostać wykonany. Wcięciem może być pojedyncza spacja lub jej wielokrotność. Natomiast **wcięciem nie może być tabulacja**. Ważne, żeby liczba spacji tworzących wcięcia była taka sama, za każdym razem, gdy robimy je na jednym poziomie. Wygodnie jest, gdy edytor, którego używasz, potrafi wstawiać spacje, gdy naciśniesz klawisz tabulatora. W moim edytorze kodu (Geany), właśnie tak jest i mam ustawione wcięcia na dwa znaki spacji. W przykładach zawartych w tym podręczniku będę stosował także takie wcięcia. Dokumentacja Nima zaleca stosowanie dokładnie takiej liczby spacji tworzących wcięcia.

Gdy program ma sprawdzić jeden warunek, używamy `if`. Jeśli program ma do sprawdzenia kilka warunków, użyjemy wtedy konstrukcji `if` i `elif`. W momencie, gdy jeden z nich jest spełniony, pozostałe nie są wówczas sprawdzane. Z `else` korzystamy wówczas, gdy program ma zrealizować jakiś kod, gdy żaden z powyższych warunków nie został spełniony.

Oto przykład programu określającego czy osoba posiadająca dany wiek jest pełnoletnia, czy nie:

```
let age = 17
if age < 18:
  echo "Nie jesteś pełnoletni."
else:
  echo "Jesteś pełnoletni."
```

```
Nie jesteś pełnoletni.
```

Kolejny przykład. Tym razem wykorzystamy `if`, `elif` i `else`:

```
let satellite = "Fobos"

if satellite == "Deimos":
  echo "Deimos jest mniejszym z dwóch ksieżyców Marsa."
elif satellite == "Fobos":
  echo "Fobos jest większym z dwóch księżyców Marsa."
else:
  echo "Mars nie posiada takiego księżyca..."
```

```
Fobos jest większym z dwóch księżyców Marsa.
```

W powyższym przykładzie został użyty podwójny znak równości `==` będący operatorem porównania. Korzystamy z niego wówczas, gdy chcemy sprawdzić, czy jedna wartość jest równa drugiej. Z kolej pojedynczy znak równości `=` jest tzw. operatorem przypisania. Korzystamy z niego, gdy np. do zmiennej chcemy przypisać wartość.

Poniżej tabela zawierająca zestawienie operatorów porównania:

Operator	Opis
==	równa się
!=	nie równa się
<	mniejsze niż
>	większe niż
<=	mniejsze lub równe
>=	większe lub równe

Gdy np. do sprawdzenia mamy więcej niż jeden warunek, wówczas możemy skorzystać z operatorów logicznych. Operatory logiczne służą do testowania prawdziwości wyrażenia składającego się z jednej lub więcej wartości logicznych. Mamy do dyspozycji następujące operatory logiczne: `and` (i), `or` (lub), `xor` (alternatywa wykluczająca), `not` (nie jest).

- logiczne `and` zwraca `true` tylko wówczas, gdy wszystkie elementy biorące udział w teście zwracają `true`,
- logiczne `or` zwraca `true` jeśli chociaż jeden element biorący udział w teście zwraca `true`,
- logiczne `xor` zwraca `true` jeśli jeden element zwraca `true` a pozostałem nie,
- logiczne `not` neguje prawdziwość swojego elementu zmieniając `true` na `false` lub odwrotnie.

Przykład wykorzystania operatora logicznego `and` — kalkulator emerytalny:

```
let
  sex = "mężczyzna"
  age = 65

if sex == "mężczyzna" and age >= 65:
  echo "Brawo! możesz iść na emeryturę."
elif sex == "kobieta" and age >= 60:
  echo "Brawo! Możesz iść na emeryturę."
else:
  echo "Niestety, nie możesz jeszcze iść na emeryturę."
```

```
Brawo! Możesz iść na emeryturę.
```

Jeśli mamy więcej warunków do sprawdzenia można użyć wygodniejszego sposobu — instrukcji `case` i `of`:

```
let pizza = "clasic"

case pizza:
  of "margharita":
    echo "sos, ser, oregano"
  of "clasic":
    echo "sos, ser, salami, pieczarki, szynka, oregano"
  of "romana":
    echo "sos, ser, kurczak, cebula, groszek, oregano"
```

```
sos, ser, salami, pieczarki, szynka, oregano
```

W powyższym przykładzie po `case pizza:` został wydzielony blok tekstu dla wszystkich przypadków opisanych przez instrukcje `of`. Nie ma konieczności tworzenia takiego bloku. W następnych przykładach nie będę już go stosował.

Kolejny przykład przedstawia możliwość użycia przedziałów liczbowych dla instrukcji `case` i `of`. Sprawdźmy, czy liczba całkowita jest dodatnia, ujemna, czy równa zero:

```
let number = -2

case number
of low(int).. -1:
  echo "Liczba ujemna"
of 0:
  echo "Zero"
of 1..high(int):
  echo "Liczba dodatnia"
```

```
Liczba ujemna
```

Poniższy kod przedstawia sposób zastąpienia konstrukcji `if`, `elif` z operatorem logicznym `or` instrukcją `case`:

```
let num = 5

# Wersja z if
if num == 2 or num == 3 or num == 5:
  echo "To jest liczba pierwsza."
elif num == 4 or num == 6 or num == 8:
  echo "To jest liczba złożona."
elif num == 0 or num == 1:
  echo "To nie jest ani liczba pierwsza ani liczba złożona."
else:
  echo "Nieprawidłowa wartość."

# Wersja z case
case num
of 2, 3, 5:
  echo "To jest liczba pierwsza."
of 4, 6, 8:
  echo "To jest liczba złożona."
of 0, 1:
  echo "To nie jest ani liczba pierwsza ani liczba złożona."
else:
  echo "Nieprawidłowa wartość."
```

Wynikiem obu wersji programów będzie `To jest liczba pierwsza.`

Do dyspozycji mamy różne sposoby wykorzystania instrukcji warunkowych. Tylko od nas zależy, która konstrukcja będzie wygodniejsza do realizacji naszego zadania.

6 Funkcje

Funkcje są wydzielonymi częściami kodu i odgrywają rolę podprogramów. Do funkcji możemy przekazywać argumenty, a ona może, choć nie musi, zwracać wartość.

Funkcje pozwalają na podział programu na logiczne części, dzięki czemu możemy wygodniej zarządzać kodem źródłowym.

Do zdefiniowania funkcji użyjemy polecenia `proc`, a następnie podamy nazwę funkcji zakończoną nawiasami okrągłymi (w następnych przykładach nawiasy te się nam przydadzą). Na końcu umieszczamy znak `=`, który otwierać będzie blok kodu do wykonania. Funkcję możemy wywołać, podając jej nazwę wraz z nawiasami.

Na początek stwórzmy prostą funkcję, która będzie wyświetlała napis `Hello world!`. Spójrz, proszę na poniższy przykład:

```
proc hello() =          # definicja
  echo "Hello world!" # funkcji

hello()               # wywołanie funkcji
```

```
Hello world!
```

Stwórzmy teraz funkcję, do której przekażemy argument. W nawiasach po nazwie funkcji wpiszemy nazwę zmiennej, pod którą będzie przechowywana wartość wewnątrz funkcji. Musimy również podać, jakiego typu planujemy przekazać wartość. W naszym przykładzie będzie to `string`:

```
proc hi(name: string) =
  echo "Witaj ", name

hi("Adam")
```

```
Witaj Adam
```

Oba powyższe przykłady przedstawiały funkcje, które nie zwracały żadnych wartości — służyły tylko do wyświetlania napisu. Teraz stworzymy funkcję, która taką wartość będzie zwracała, dzięki czemu będziemy ją mogli przypisać do zmiennej:

```
proc sum(x: int, y: int): int =
  return x + y

var result = sum(3, 4)

echo result
```

```
7
```

Do powyższej funkcji przekazujemy dwie wartości, które w funkcji są przechowywane pod zmiennymi `x` i `y`, obie typu całkowitego `int`. Wartość tego samego typu będzie zwracana przez naszą funkcję. Polecenie `return` służy właśnie do zwracania wartości.

Może pojawić się sytuacja, że dla funkcji wymagającej argumentów nie zostaną one podane. Wówczas program zostanie przerwany. Aby temu zaradzić, możemy ustalić domyślne wartości dla zmiennych w funkcji. Oto przykład:

```
proc sum(x: int = 0, y: int = 0): int =
  return x + y}

var result = sum()

echo result
```

```
0
```

Wywołując powyższą funkcję, nie podano żadnych argumentów. W efekcie, do obliczenia sumy zostały wykorzystane wartości domyślne dla `x` i `y` wynoszące 0.

7 Moduły

Moduły są to programy, które zawierają definicje funkcji. Funkcje takie usprawniają programowanie a dzięki temu, że znajdują się w swoich modułach (oddzielnych plikach), są odseparowane od właściwego kodu programu.

7.1 Instalacja i import modułów

Niektóre moduły są instalowane wraz z Nimem. Jeśli chcemy z nich skorzystać, wystarczy je zaimportować. Są jednak moduły dodatkowe, z których, kiedy chcemy skorzystać, trzeba je najpierw zainstalować.

Do instalacji dodatkowych modułów służy menedżer pakietów o nazwie `nimble`.

Zanim jednak będziemy chcieli wykonać jakieś operacje, warto sobie najpierw zsynchronizować listę modułów znajdującą się na naszym komputerze z tą, na serwerze:
`nimble refresh`

Wiec, gdy chcemy zainstalować nowy moduł:
`nimble install nazwa_modułu`

Dla przykładu, żeby zainstalować moduł o nazwie `Pixie` obsługujący grafikę, w terminalu wydajemy polecenie:
`nimble install pixie`

Gdy jednak chcemy odinstalować moduł:
`nimble uninstall nazwa_modułu`

Możemy także wyświetlić listę dostępnych do instalacji modułów.
`nimble list`

Jeśli jednak wiemy jakiego modułu szukamy to, zamiast pobierać informację o wszystkich, możemy wyszukać konkretnego:
`nimble search nazwa_modułu`

W ramach krótkiego ćwiczenia napiszmy krótki program, który obliczy sinus kąta, wyrażonego w radianach. Do tego celu wykorzystamy funkcję `sin()` z modułu `math`, który wcześniej zaimportujemy. Jeśli moduł, który chcemy zaimportować pochodzi ze standardowej instalacji Nima, to dobrym zwyczajem jest (choć nie jest to niezbędne) dopisanie przed nazwą modułu `std`:

```
import std/math

echo PI
echo sin(PI)
```

```
3.141592653589793
1.224646799147353e-16
```

Moduł `math` oprócz definicji funkcji `sin()` posiada również zdefiniowaną wartość liczby `PI`, która została wyświetlona w powyższym przykładzie.

Oczywiście `sin(π)` wynosi `0`, jednak dokładność liczby π użyta w tym przykładzie, nie pozwala osiągnąć takiej wartości przy takiej precyzji wyświetlanego wyniku. Nie musimy się teraz tym przejmować.

Jeśli chcemy wykorzystać tylko jedną funkcję z danego modułu, wówczas możemy zaimportować tylko tę funkcję:

```
from std/math import sin
```

Jeśli zajdzie taka potrzeba można importować moduły z wykluczeniem konkretnych funkcji. Dla przykładu zaimportujmy moduł `math` ale bez funkcji `PI`:

```
import std/math except PI
```

Po zaimportowaniu modułu mamy bezpośredni dostęp do zdefiniowanych w nim funkcji. Możemy się do nich również odwoływać, poprzedzając ich nazwę nazwą modułu oddzieloną kropką o nazwy funkcji. Oto przykład:

```
import std/math

echo math.sqrt(9.0)
```

Można również stosować tzw. aliasy (`as`), dzięki którym będziemy mogli przechowywać zaimportowany moduł pod własną nazwą.

```
import std/math as m_functions

echo m_functions.sqrt(9.0)
```

Korzystanie z odpowiednich modułów przyspiesza pracę nad kodem i sprawia, że programuje się wygodniej. W dalszej części podręcznika skupimy się bardziej na kilku modułach, do których częściej będziemy sięgać.

7.2 Własne moduły

Istnieje możliwość tworzenia własnych modułów. Wymaga to jednak wcześniejszego poznania wiadomości dotyczących funkcji.

Nasz moduł, żeby można było z niego skorzystać, podczas kompilacji powinien znajdować się w tym samym folderze, co nasz główny program.

Prześledźmy poniższy przykład:
Tworzymy plik o nazwie `hi_module.nim`. W tym pliku tworzymy funkcję o nazwie `hi`, która będzie służyła do wyświetlania napisu `Hello world`. Oto zawartość pliku `hi_module.nim`:

```
proc hi*() =
  echo "Hello world."
```

Gwiazdka (`*`) umieszczona po nazwie funkcji a przed nawiasem pozwala na eksport tej funkcji. Gdybyśmy jej nie użyli, nie moglibyśmy tej funkcji zaimportować w głównym programie.

Zawartość głównego programu i wynik jego działania:

```
import hi_module

hi()
```

```
Hello world.
```

W kolejnym przykładzie stworzymy moduł, zawierający funkcję `sum`, sumującą dwie liczby całkowite przekazane do funkcji, jako argumenty. Tworzymy zatem plik główny `main.nim` oraz plik modułu o nazwie `calculate.nim` a wewnątrz tego pliku tworzymy funkcję `sum`:

```
proc sum*(a: int, b: int): int =
  return a + b
```

Zawartość głównego pliku programu `main.nim`:

```
import calculate

echo sum(3, 4)
```

```
7
```

8 Sekwencje i moduł sequtils

8.1 Sekwencje

Sekwencje pozwalają przechowywać wiele wartości, przy czym nie trzeba z góry znać liczby tych wartości. Każda **sekwencja może przechowywać wartości tylko jednego typu**.

Stwórzmy sekwencję i przypiszmy do niej wartości:

```
var x = @[1, 2, 3]

echo x
echo x.type
```

```
@[1, 2, 3]
seq[int]
```

Możemy także zadeklarować sekwencję bez dodawania do niej żadnych wartości:

```
var x = newSeq[string]()

echo x
echo x.type
```

```
@[]
seq[string]
```

Zadeklarujmy nową sekwencję i dodajmy do niej jakieś wartości:

```
var x = newSeq[string]()
x.add("hello")
x.add("world")

echo x
```

```
@["hello", "world"]
```

Możemy też usunąć element sekwencji, gdy znamy jego indeks:

```
var x = @["hello", "world"]

echo x

x.del(1)

echo x
```

```
@["hello", "world"]
@["hello"]
```

W przypadku sekwencji możemy użyć takich samych konstrukcji, jakie zostały przedstawione dla tekstów w pierwszym przykładzie na stronie 14. Wówczas będziemy się odwoływać już nie do pojedynczych znaków, ale do całych elementów, z których składa się sekwencja.

8.2 Moduł sequtils

W module o nazwie `sequtils` znajduje się więcej funkcji służących do obsługi sekwencji.

Po zaimportowaniu modułu możemy połączyć kilka sekwencji w jedną, wykorzystując funkcję `concat()`, gdzie jako argumenty tej funkcji podajemy kolejne sekwencje:

```
import sequtils

var
  x = @[1, 2]
  y = @[3, 4]
  z = @[5, 6]
  c = concat(x, y, z)

echo c
```

```
@[1, 2, 3, 4, 5, 6]
```

Do dyspozycji mamy również funkcję o nazwie `count()` służąca do zliczania liczby wystąpień danego elementu sekwencji. Zliczmy, ile jest jedynek w poniższej sekwencji:

```
import sequtils

var x = @[0, 1, 1, 2, -1]

echo x.count(1)
```

```
2
```

Sprawdźmy, na jakiej pozycji stoją minimalne i maksymalne wartości sekwencji:

```
import sequtils

var x = @[5, 3, -1, 2, 5]

echo x.minIndex
echo x.maxIndex
```

```
2
0
```

Jeśli maksymalnych lub minimalnych wartości jest więcej niż jedna, to funkcje `minIndex()` i `maxIndex()` zwracają indeksy pierwszych napotkanych wartości.

Wyświetlmy teraz wartości odpowiadające tym indeksom:

```
import sequtils

var x = @[5, 3, -1, 2, 5]

echo x[x.minIndex]
echo x[x.maxIndex]
```

```
-1
5
```

Funkcja `deduplicate()` służy do usunięcia duplikatów wartości sekwencji:

```
import sequtils

var x = @[1, 2, 2, 3, 3, 3]

echo x.deduplicate
```

```
@[1, 2, 3]
```

Z sekwencji możemy pobierać wartości, które spełniają określony przez nas warunek. Do tego służy funkcja `filter()`. Zatem użyjemy filtra do wyświetlenia tylko tych wartości sekwencji, które są mniejsze niż 3:

```
import sequtils

var x = @[-4, 7, 0, 3]

echo x.filter(proc(i: int): bool = i < 3)
```

```
@[-4, 0]
```

W powyższym przykładzie została użyta jednolinijkowa funkcja `proc()` sprawdzająca warunek i zwracająca wartość logiczną (`true` lub `false`), dzięki czemu funkcja `filter()` mogła zwrócić tylko te wartości, które spełniały ten wewnętrzny warunek.

Funkcja `keepIf()` daje taki sam rezultat co `filter()` z tą różnicą, że zmiany są zapisywane w aktualnej sekwencji:

```
import sequtils

var x = @[-4, 7, 0, 3]

x.keepIf(proc(i: int): bool = i < 3)

echo x
```

```
@[-4, 0]
```

Istnieje także możliwość podziału jednej sekwencji na kilka. Wówczas wewnątrz sekwencji są tworzone mniejsze, zawierające dane z pierwotnej sekwencji:

```
import sequtils

var x = @[1, 2, 3, 4, 5, 6, 7, 8]

echo x.distribute(2)
```

```
@[@[1, 2, 3, 4], @[5, 6, 7, 8]]
```

W celu zwielokrotnienia sekwencji użyjemy funkcji `cycle()`, dla której argumentem jest liczba powtórzeń. Dla przykładu skopiujmy trzykrotnie wartości sekwencji:

```
import sequtils

var x = @[1, 2, 3]

echo x.cycle(3)
```

```
@[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Dzięki funkcji `insert()` wstawimy wartości w miejsce o podanym indeksie. Funkcja przyjmuje dwa argumenty, gdzie jako pierwszy podajemy wartość, którą chcemy wstawić, a jako drugi numer indeksu, pod którym ta wartość ma zostać wstawiona. Dla przykładu wstawmy wartość 4 na pozycję o indeksie 2:

```
import sequtils

var x = @[1, 2, 3]
x.insert(4, 2)

echo x
```

```
@[1, 2, 4, 3]
```

Gdybyśmy chcieli zmienić zawartość sekwencji, np. do każdej wartości dodać jakąś wartość, wówczas możemy skorzystać z funkcji `map()`. Dla przykładu pomnóżmy każdą wartość w sekwencji przez dwa:

```
import sequtils

var x = @[0, 1, 2, 3]
echo x.map(proc(x: int): int = 2*x)

echo x
```

```
@[0, 2, 4, 6]
```

9 Pętle

Pętle są konstrukcjami, które pozwalają na wielokrotne powtarzanie szeregu instrukcji w nich zawartych. W Nim mamy do dyspozycji dwa rodzaje pętli — `while` i `for`. Obie pętle mimo tego, że służą do tego samego — powtarzają część kodu — działają na różne sposoby.

9.1 Pętla while

Jest to pętla, która sterowana jest warunkiem. Wykonywać się będzie dopóty, dopóki warunek ten jest spełniany. Nie ma potrzeby określania na początku, ile razy pętla musi się wykonać. Dlatego pętla ta znajduje zastosowanie głównie w sytuacjach, kiedy nie jesteśmy w stanie na początku określić liczby powtórzeń.

Dla pętli `while` możemy przekazać informację, że warunek jest spełniony. Wówczas pętla staje się pętlą nieskończoną:

```
while true:
  echo "Hello world!"
```

Wynikiem działania powyższego kodu będzie nieskończone wyświetlanie tekstu `Hello world!`.

Pętla `while` z licznikiem:

```
var x = 1

while x <= 5:
  echo x
  x = x + 1
```

```
1
2
3
4
5
```

Zwiększanie lub zmniejszanie wartości zmiennej możemy realizować na kilka sposobów.

Zwiększanie lub zmniejszanie wartości zmiennej możemy realizować na kilka sposobów. Oto przykłady:

```
var x = 1

# zwiększanie wartości
x = x + 1
x += 1
inc x # można zwiększać wartość tylko o 1

# zmniejszanie wartości
x = x - 1
x -= 1
dec x # można zmniejszać wartość tylko o 1
```

Zwiększanie wartości zmiennej dokładnie o 1 nazywa się **inkrementacją**. Analogicznie, zmniejszanie wartości dokładnie o 1 nazywa się **dekrementacją**.

Instrukcja `continue` wznawia działanie pętli, jednocześnie zaprzestając wykonywania dalszych instrukcji dla danego powtórzenia:

```
var x = 0

while x < 5:
  inc x
  if x == 3:
    continue
  echo x
```

```
1
2
4
5
```

Instrukcja `break` przerywa działanie pętli, w której została wywołana:

```
var x = 1

while true:
  echo x
  if x == 3:
    break
  inc x
```

```
1
2
3
```

9.2 Pętla for

Pętla `for` sprawdza się w sytuacjach, kiedy z góry wiemy, ile razy ta pętla ma się wykonać. Pętla `for` posiada wbudowany licznik, który możemy wykorzystać do odmierzania liczby

powtórzeń.

Funkcja `countup()` może przyjąć trzy argumenty, kolejno: liczba początkowa, liczba końcowa, krok. Trzeci argument, czyli krok, jest opcjonalny. Tzn. nie musimy go podawać, gdy wartość ma się zmieniać o jeden — taka właśnie wartość jest ustawiona domyślnie.

```
for i in countup(1, 3, 1):
  echo i
```

```
1
2
3
```

Zmienna `i` w powyższym przykładzie jest to tzw. iterator pętli (łac. iteratio, itero — powtarzanie) — jest to zmienna sterująca działaniem pętli. Przyjmuje ona wartości zawarte w przedziale, określonym przez liczbę początkowa i liczba końcową. Obie te liczby należą do tego przedziału.

Analogicznie działa funkcja `countdown()` odliczająca od największej wartości do najmniejszej:

```
for i in countdown(3, 1, 1):
  echo i
```

```
3
2
1
```

Pętla ta dobrze sprawdza się podczas wyświetlania zawartości sekwencji. Tym razem będziemy iterować po jej poszczególnych elementach:

```
var colors = @["zielony", "czerwony", "niebieski", "biały"]

for color in colors:
  echo color
```

```
zielony
czerwony
niebieski
biały
```

Można także wyświetlić zawartość sekwencji wraz z numerami indeksów, pod którymi znajdują się jej elementy:

```
var colors = @["zielony", "czerwony", "niebieski", "biały"]

for index, color in colors:
  echo index, ". ", color
```

```
0. zielony
1. czerwony
2. niebieski
3. biały
```

W powyższym przykładzie na pierwszym miejscu stoi zmienna `index`, która pobiera kolejne numery indeksu. Nazwa tej zmiennej jest dowolna. Na drugim miejscu stoi zmienna `color`, która przyjmuje kolejne elementy sekwencji. Nazwa tej zmiennej także jest dowolna. Na końcu, po `in` znajduje się nazwa wyżej zadeklarowanej sekwencji.

10 Tablice

Tablice pozwalają przechowywać wiele wartości. Podobnie, jak sekwencje, muszą zawierać elementy jednego typu. To, co odróżnia tablice od sekwencji, to szybkość. Operacje wykonywane na tablicach z reguły są szybsze niż te wykonywane na sekwencjach. To jednak ma też swoje konsekwencje — wielkość tablicy musi zostać określona przed kompilacją programu. Czego nie musieliśmy robić w przypadku sekwencji.

W pierwszym przykładzie stwórzmy tablicę znakową, zawierającą trzy litery:

```
var arr = ['a', 'b', 'c']

echo arr.type
echo arr
```

```
array[0..2, char]
['a', 'b', 'c']
```

Dzięki funkcji `type()` uzyskaliśmy informacje o indeksach, pod którymi są przechowywane wartości oraz o typie danych przechowywanych przez tablicę.

Do poszczególnych elementów tablicy odwołujemy się tak samo, jak w przypadku sekwencji, podając numer indeksu, pod którym znajduje się dany element. Warto przypomnieć, że numery indeksów rozpoczynają się od 0.

```
var arr = ['a', 'b', 'c']

echo arr[0]
echo arr[^1]
```

```
a
c
```

Do wyświetlania zawartości tablicy często wygodnie jest wykorzystać pętlę `for`:

```
var arr = ['a', 'b', 'c']

for char in arr:
  echo char
```

```
a
b
c
```

Jeśli w chwili kompilacji programu nie mamy jeszcze danych, które możemy wstawić do tablicy, wówczas możemy ją zadeklarować a uzupełnić podczas realizacji programu.

```
var arr: array[3, int]

echo arr.type
echo arr
```

```
array[0..2, int]
[0, 0, 0]
```

Podczas deklaracji tablicy, tak jak już wcześniej zostało powiedziane, musimy podać, jak duża ma być tablica oraz jakiego typu dane będą w niej przechowywane. Jak możemy zaobserwować w powyższym przykładzie, deklaracja tablicy bez dodawania do niej wartości, powoduje uzupełnienie tablicy zadeklarowaną wcześniej liczbą następujących wartości:

- 0, w przypadku tablicy przechowującej liczby całkowite,
- `0.0`, w przypadku tablicy przechowującej liczby zmiennoprzecinkowe,
- `""`, w przypadku tablicy przechowującej ciągi znaków,
- `'\x00'`, w przypadku tablicy przechowującej pojedyncze znaki
- `false`, w przypadku tablicy przechowującej wartości logiczne.

Można także rozszerzyć deklarację tablicy o dodanie do niej wartości:

```
var arr1: array[3, int] = [1, 2, 3]
var arr2 = [1, 2, 3]

echo arr1.type
echo arr1
echo arr2.type
echo arr2
```

```
array[0..2, int]
[1, 2, 3]
array[0..2, int]
[1, 2, 3]
```

Oba sposoby stworzenia tablic (`arr1` i `arr2`) dają ten sam efekt.

Dodajmy teraz jakieś wartości do zadeklarowanej tablicy:

```
var names: array[3, string]
names[0] = "Adam"
names[1] = "Ewa"
names[2] = "Michał"

echo names
```

```
["Adam", "Ewa", "Michał"]
```

W dotychczasowych przykładach wykorzystywaliśmy tylko tablice jednowymiarowe. Teraz stwórzmy tablicę zawierającą dwie tablice, które będą miały po dwie liczby całkowite. Poniższy przykład pokazuje, w jaki sposób odwoływać się do poszczególnych indeksów tych tablic:

```
var x: array[2, array[2, int]] = [
                                  [1, 2],
                                  [3, 4]
                                 ]

echo x
echo x[0][0]
echo x[0][1]
echo x[1][0]
echo x[1][1]
```

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

Jeśli tablicę uzupełniamy danymi przed kompilacją, wówczas nie musimy podawać szczegółów dotyczących jej rozmiaru i typu przechowywanych danych — są one pobierane na podstawie wprowadzonych wartości. Zatem zapiszmy ten sam przykład, co powyżej, w krótszej formie:

```
var x = [
         [1, 2],
         [3, 4]
        ]

echo x
echo x[0][0]
echo x[0][1]
echo x[1][0]
echo x[1][1]
```

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

11 Tuple

Tuple, zwane inaczej krotkami pozwalają na przechowywanie wielu danych, jednak w odróżnieniu od sekwencji i tablic, pozwalają przechowywać dane różnych typów.

Oto przykładowa tupla, zawierająca dwie wartości — tekst i liczbę całkowitą:

```
var tup: (string, int) = ("Adam", 15)

echo tup
echo tup.type
```

```
("Adam", 15)
(string, int)
```

Jak można zauważyć w powyższym przykładzie, tupla jest reprezentowana przez nawiasy okrągłe.

Gdy tworząc krotkę, uzupełniamy ją od razu wartościami, wówczas nie musimy podawać typów danych — typy będą pobrane z wprowadzonych danych:

```
var tup = ("Adam", 15)

echo tup
echo tup.type
```

```
("Adam", 15)
(string, int)
```

Możemy się odwoływać do poszczególnych elementów tupli, podobnie, jak w przypadku sekwencji i tablic, podając numer indeksu:

```
var tup = ("Adam", 15)

echo tup[0]
echo tup[1]
```

```
Adam
15
```

Możemy także określić, gdzie dokładnie mają być przechowywane dane. Wyświetlmy od razu poszczególne elementy, tak, jak to robiliśmy do tej pory i drugim sposobem — wywołując wartości po nazwach, pod którymi przechowywane są one przechowywane:

```
var tup: tuple[name: string, age: int] = ("Adam", 15)

echo tup
echo tup[0]
echo tup[1]
echo tup.name
echo tup.age
```

```
(name: "Adam", age: 15)
Adam
15
Adam
15
```

Zadeklarujmy teraz krotkę, ale nie uzupełniajmy jej od razu, wartości dodamy już po jej utworzeniu:

```
var planet: tuple[name: string, water: bool]

planet.name = "Ziemia"
planet.water = true

echo planet
```

```
(name: "Ziemia", water: true)
```

Jest jeszcze jeden sposób deklarowania tupli uważany za bardzo czytelny. Użyjemy instrukcji `type`, dzięki której będziemy mogli stworzyć własne typy danych:

```
type
  Tup_template = tuple
    name: string
    age: int

var person: Tup_template = ("Ewa", 20)

echo person
echo person.name
echo person.age
```

```
(name: "Ewa", age: 20)
Ewa
20
```

12 Zbiory i moduł sets

12.1 Zbiory

Zbiór (set) zwany inaczej zestawem danych, to kolejna konstrukcja służąca do przechowywania danych. Ogólnie zbiór jest komputerową implementacją matematycznej koncepcji zbioru skończonego. Przypomina on trochę tablicę, z tą różnicą, że zbiór może przechowywać tylko unikatowe (niepowtarzające się) wartości. Kolejność przechowywanych elementów nie jest śledzona, więc nie możemy użyć indeksu, aby uzyskać wartość.

Zbiory w podstawowej wersji mogą zawierać następujące typy danych: `char`, `int8`, `int16`, `enum`, `uint8` / `byte-uint16`.

Stwórzmy nasz pierwszy zbiór:

```
let letters: set[char] = {'b', 'a', 'c'}

echo letters
```

```
{'a', 'b', 'c'}
```

Proszę zwrócić uwagę na kolejności znaków w kodzie i w wyniku. Tak, jak już wyżej wspomniałem, zbiory nie pilnują kolejności danych.

Oczywiście, jeśli zbiory od razu uzupełniamy danymi, to powyższy zapis możemy uprościć:

```
let letters = {'b', 'a', 'c'}
```

Wykorzystując zbiory, możemy pozbywać się duplikatów danych:

```
let letters = {'a', 'a', 'b', 'c'}
```

```
{'a', 'b', 'c'}
```

Do elementów zbiorów nie możemy odwoływać się po indeksach, ponieważ ich nie posiadają, ale możemy sprawdzić, czy dany element jest w zbiorze, wykorzystując polecenie `in`:

```
let letters: set[char] = {'a', 'b', 'c'}

echo 'a' in letters
echo 'z' in letters
```

```
true
false
```

Analogicznie możemy sprawdzić czy jakiś element nie występuje w zbiorze. Użyjemy do tego polecenia `notin`:

```
let letters: set[char] = {'a', 'b', 'c'}

echo 'a' notin letters
echo 'z' notin letters
```

```
false
true
```

Możemy tworzyć zbiory i wypełniać je danymi podając przedziały wartości, które mają znaleźć się w tych zbiorach:

```
let letters: set[char] = {'a' .. 'z'}

echo letters
```

```
{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',
'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x',
'y', 'z'}
```

Nowe przedziały możemy podawać po przecinku:

```
let letters: set[char] = {'a' .. 'z', 'A' .. 'Z'}

echo letters
```

```
{'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L',
'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v',
'w', 'x', 'y', 'z'}
```

Gdybyśmy chcieli utworzyć zbiór z wartościami liczbowymi (`uint8`), np. od 0 do 15 wówczas zapiszemy to w następujący sposób:

```
let numbers: set[uint8] = {0.uint8 .. 15.uint8}

echo numbers
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
```

Do sprawdzenia długości zbioru możemy użyć funkcji `len()` lub `card()`. Obie funkcje dają te same efekty:

```
let letters: set[char] = {'a' .. 'z'}

echo len(letters)
echo card(letters)
```

```
26
26
```

W rzeczywistości `card()` jest tylko aliasem (inną nazwą) dla funkcji `len()`.

12.2 Moduł sets

Zbiory posiadają o wiele większe możliwości, jeśli skorzystamy z modułu `std/sets`.

Dzięki funkcji `toHashSet()` mamy możliwość przekonwertowania np. tablicy lub sekwencji do zbioru:

```
import std/sets

let x = toHashSet([1, 2, 3, 4])

echo x
echo x.type
```

```
{3, 4, 2, 1}
HashSet[system.int]
```

Funkcja `toHashSet()` tworzy zbiór, który może przechowywać tylko unikatowe wartości i nadal nie pilnuje on kolejności danych. Jednak różnicą w stosunku do podstawowej wersji zbioru jest możliwość przechowywania większej liczby typów danych:

```
import std/sets

let x = toHashSet(["Hello", "world", "!"])

echo x
echo x.type
```

```
{"Hello", "world", "!"}
HashSet[system.string]
```

W module `std/sets` znajdziemy funkcję `toOrderedSet()`, dzięki której będziemy mogli tworzyć zbiory uporządkowane. Jednak korzystanie z uporządkowanych zbiorów zajmuje więcej czasu niż korzystanie ze zbiorów nieuporządkowanych:

```
import std/sets

let x = toOrderedSet([1, 2, 3])
let y = toHashSet([1, 2, 3])

echo x
echo y
```

```
{1, 2, 3}
{3, 2, 1}
```

Na zbiorach możemy wykonywać operacje matematyczne:

```
import std/sets

let
  x = toHashSet([1, 2, 3, 4, 5])
  y = toHashSet([0, 2, 4, 6, 8])
  z1 = x+y            # suma
  z2 = x*y            # iloczyn (część wspólna)
  z3 = x-y            # różnica
  z4 = x -+- y        # symetryczna różnica
  z5 = x == y         # porównanie


echo "Suma: ", z1
echo "Iloczyn: ", z2
echo "Różnica: ", z3
echo "Symetryczna różnica: ", z4
echo "Porównanie: ", z5
```

```
Suma: {3, 4, 2, 6, 5, 8, 0, 1}
Iloczyn: {4, 2}
Różnica: {5, 3, 1}
Symetryczna różnica: {3, 6, 5, 8, 0, 1}
Porównanie: false
```

W przypadku sumy, zbiór wynikowy będzie zawierał wszystkie elementy zawarte w obu sumowanych zbiorach (duplikaty wartości będą usuwane).

Iloczyn pozostawi tylko te wartości, które znajdują się zarówno w pierwszym, jak i w drugim zbiorze.

Różnica pozostawi tylko te wartości, które znajdują się w pierwszym zbiorze i nie ma ich w drugim.

Symetryczna różnica pozostawi wszystkie te wartości, które znajdują się tylko w pierwszym zbiorze i nie ma ich w drugim oraz wszystkie te wartości, które znajdują się tylko w drugim zbiorze i nie ma ich w pierwszym.

Porównanie zwróci wartość logiczną — `true` lub `false`.

Dzięki funkcji `clear()` możemy usunąć zawartość zbioru:

```
import std/sets

var x = toHashSet([2, 3, 6])

echo x

clear x

echo x
```

```
{3, 2, 6}
{}
```

Możemy sprawdzić czy dany element znajduje się w zbiorze także w inny sposób niż ten, wykorzystujący polecenie `in`. Skorzystamy z funkcji `contains()`:

```
var x = {1, 2, 3}

echo x.contains(0)
echo x.contains(2)
```

```
false
true
```

Jeśli nie ma to dodaje...

```
import std/sets

var x = toHashSet([1, 2, 3])

echo x.containsOrIncl(0)
echo x.containsOrIncl(0)
echo x.containsOrIncl(2)
```

```
false
true
true
```

13 Niestandardowe typy danych

W języku Nim możemy tworzyć własne typy danych. Możliwość ta często jest wykorzystywana do konstrukcji tupli lub obiektów. W tym punkcie opiszę sposób tworzenia własnych typów danych na podstawie deklaracji tupli.

Dla przykładu utworzymy typ danych, a potem na jego podstawie krotkę. Krotka ta będzie przechowywała znak ASCII i jego kod zapisany w systemie dziesiętnym. Tworzymy zatem nowy typ danych o nazwie `Tup_ascii` (dobrym zwyczajem jest zapisywanie nazw typów wielką literą), który będzie oparty na krotce przechowującej dane znakowe `char` i liczbę całkowitą dziesiętną `int`:

```
type
  Tup_ascii = tuple[character: char, code: int]

var letter: Tup_ascii = ('a', 97)

echo letter.type
echo letter
echo letter.character
echo letter.code
```

```
Tup_ascii
(character: 'a', code: 97)
a
97
```

Sprawdzając typ danych naszej tupli o nazwie `letter`, otrzymaliśmy po prostu nazwę typu, który zadeklarowaliśmy, czyli `tup_ascii`.

Bardziej przejrzystym sposobem definiowania nowego typu danych jest ten, który został użyty na końcu rozdziału dotyczącego tupli. Sprawdza się on zwłaszcza w przypadku, gdy mamy więcej danych do przechowania w tupli:

```
type
  Tup_ascii = tuple
    character: char
    code: int
```

14 Wyliczenia (enums)

Wyliczenia...

15 Pseudolosowość

Zdarzeniem losowym może być np. rzut kośćmi do gry lub monetą. W przypadku komputerów nie ma możliwości uzyskania zdarzeń prawdziwie losowych. Ponieważ wszystko, co się dzieje na komputerze, jest wynikiem obliczeń. Zatem podstawiając te same wartości do tego samego wzoru, otrzymamy te same wyniki. Chyba że komputer ulegnie awarii, wówczas jest szansa na pewną losowość.

Dlatego na komputerach, żeby uzyskać wyniki zbliżone do prawdziwie losowych, wykorzystuje się tzw. generatory liczb pseudolosowych. Są to programy, które potrafią obliczać wartości przypominające te, uzyskane w wyniku losowych zdarzeń. Żeby takie obliczenia dawały losowe wyniki, potrzebne jest jeszcze tzw. ziarno, czyli pewne początkowe informacje, według których będą wyznaczane liczby pseudolosowe.

Żeby móc skorzystać z liczb pseudolosowych w Nim, musimy zaimportować moduł `std/random`.

Taki generator nie utworzy liczb prawdziwie losowych. Więc wielokrotne uruchomienie poniższego programu, będzie generowało zawsze tę samą wartość:

```
import std/random

echo rand(5)
```

W powyższym przykładzie najpierw zaimportowaliśmy moduł, a potem wyświetliliśmy wylosowaną liczbę całkowitą z przedziału od 0 do 5 — obie wartości zaliczają się do losowanego przedziału.

Żeby uzyskane przez nasz program wyniki odpowiadały tym prawdziwie losowym, przed wygenerowaniem liczby musimy zainicjować nasz generator. Do inicjalizacji generatora służy funkcja `randomize()`. Wymagane jest, aby wywołać ją tylko raz, jeszcze przed pierwszym użyciem jakiejkolwiek innej funkcji z modułu `random`:

```
import std/random

randomize()

echo rand(5)
```

Od tego momentu każde uruchomienie programu wygeneruje liczbę pseudolosową podobną do wartości uzyskanej ze zdarzenia prawdziwie losowego. Tak, jak w zdarzeniach prawdziwie losowych, nie oznacza to wcale, że wylosowane liczby nie będą mogły się powtarzać.

Poniżej przedstawiam sposoby losowania liczb:

```
import std/random

randomize()

echo rand(5)         # od 0 do 5 (int)
echo rand(0..5)      # od 0 do 5 (int)
echo rand(0..<5)     # od 0 do 4 (int)
echo rand(0.0..5.0) # od 0.0 do 5.0 (float)
```

```
5
0
2
1.950257557968613
```

Do losowania elementów z gotowych zestawów danych, np. z tablic, służy funkcja `sample()`, która jako argument przyjmuje zestaw danych:

```
import std/random

randomize()

let colors = ["czerwony", "niebieski", "biały"]

echo sample(colors)
```

```
niebieski
```

Funkcja `shuffle()` pozwala zmienić kolejność (wymieszać) wartości w zbiorze danych. Robi to, nadpisując dotychczasowe dane:

```
import std/random

randomize()

var colors = ["czerwony", "niebieski", "biały"]

echo colors

shuffle(colors)

echo colors
```

```
["czerwony", "niebieski", "biały"]
["czerwony", "biały", "niebieski"]
```

Napiszmy teraz program, który wylosuje 5 liczb z przedziału od 1 do 10, ale w taki sposób, żeby wylosowane liczby nie się powtarzały. Do tego celu wykorzystamy liczby pseudoloso-

we, które będziemy dopisywali do sekwencji w pętli. Tak wylosowane wartości będą mogły się powtarzać, więc z góry nie wiemy, po ilu powtórzeniach otrzymamy wszystkie unikatowe liczby. Zatem skorzystamy z pętli `while`. Dodatkowo, po każdym losowaniu będziemy usuwali, jeśli będą występowały, powtarzające się liczby funkcją `deduplicate()`. Warunkiem wyjścia z pętli będzie osiągnięcie liczby (unikatowych) elementów sekwencji równej 5:

```
import std/random
import sequtils

randomize()

var numbers = newSeq[int]()

while len(numbers) < 5:
  numbers.add(rand(1..10))
  numbers = numbers.deduplicate

echo numbers
```

```
@[4, 7, 5, 6, 3]
```

16 Obiekty

Programowanie obiektowe jest metodą budowania programów za pomocą obiektów. Taki obiekt łączy ze sobą grupę danych, które mogą być różnych typów. Pokażę teraz, w jaki sposób tworzy się obiekty w Nim.

Obiekty i ich typy są definiowane w sekcji `type`. Bardzo podobnie było w przypadku tupli. Tworząc taką strukturę, nadajemy jej nazwę i ustawiamy jej typ na `object`. Na bazie tak zadeklarowanego obiektu możemy utworzyć jego instancję.

Na podstawie obiektu `Person` stwórzmy jego instancję — człowieka o nazwie `p1`, który będzie zawierał takie jego cechy (atrybuty), jak płeć (`sex`), imię (`name`) i wiek (`age`):

```
# Definicja typu obiektu
type
  Person = object
    sex: string
    name: string
    age: int

# Instancja typu obiektu
var p1 = Person(sex: "mężczyzna", name: "Adam", age: 15)

echo p1
echo p1.sex
echo p1.name
echo p1.age
```

```
(sex: "mężczyzna", name: "Adam", age: 15)
mężczyzna
Adam
15
```

Jeśli wartość jakiegoś atrybutu instancji obiektu ulegnie zmianie, to możemy ją zaktualizować:

```
type
  Person = object
    sex: string
    name: string
    age: int

var p1 = Person(sex: "mężczyzna", name: "Adam", age: 15)

echo p1

p1.age = 16

echo p1
```

```
(sex: "mężczyzna", name: "Adam", age: 15)
(sex: "mężczyzna", name: "Adam", age: 16)
```

Spróbujmy teraz napisać generator postaci. Program stworzy np. 5 losowych ludzi. Każda osoba będzie charakteryzować się imieniem (`name`), wiekiem (`age`) oraz profesją (`profession`). Żeby skorzystać z losowości w Nim, użyjemy modułu `std/random`. Z kolei, żeby w wygodny sposób zebrać wartości atrybutów poszczególnych instancji, skorzystamy z modułu `std/strformat`.

```
import std/random
import std/strformat

# Definicja typu obiektu
type
  Person = object
    name: string
    profession: string
    age: int

# Inicjalizacja generatora liczb losowych
randomize()

# Utworzenie tablic oraz deklaracja sekwencji
var
  names = ["Adam", "Ewa", "Wojciech", "Piotr", "Alicja"]
  professions = ["aktor", "nauczyciel", "fryzjer", "lekarz"]
  persons = newSeq[Person]()

# Pętla generująca instancje obiektów
# i zapisująca je do sekwencji
for i in countup(0, 4):
  let
    name = sample(names)
    profession = sample(professions)
    age = rand(18..90)
    p = Person(name: name, profession: profession, age: age)
  persons.add(p)

# Pętla wyświetlająca instancje obiektów z sekwencji
for person in persons:
  let str_result = &"Imię: {person.name},"&
                   &"zawód: {person.profession},"&
                   &"wiek: {person.age}."
  echo str_result
```

```
Imię: Ewa, zawód: nauczyciel, wiek: 52.
Imię: Adam, zawód: aktor, wiek: 39.
```

```
Imię: Ewa, zawód: fryzjer, wiek: 56.
Imię: Alicja, zawód: lekarz, wiek: 53.
Imię: Ewa, zawód: lekarz, wiek: 85.
```

17 Praca z plikami tekstowymi

17.1 Odczyt danych z pliku

Stwórzmy plik tekstowy o nazwie `plik.txt` i umieśćmy w nim trzy linijki tekstu. Oto zawartość pliku tekstowego:

```
To jest pierwszy wiersz.
To jest drugi wiersz.
Koniec.
```

Plik z tekstem zapisujemy w tym samym katalogu, w którym znajdować się będzie plik z naszym programem.

Teraz z pomocą krótkiego kodu odczytamy jego zawartość i wyświetlimy w terminalu:

```
let text = readFile("plik.txt")

echo text
```

```
To jest pierwszy wiersz.
To jest drugi wiersz.
Koniec.
```

Do odczytu całej zawartości pliku tekstowego posłużyliśmy się funkcją o nazwie `readFile()`, dla której argumentem jest nazwa pliku lub pełna ścieżka do pliku, jeśli ten znajduje się w innym katalogu niż program. Zawartość pliku tekstowego została przypisana do stałej o nazwie `text` i wyświetlona w terminalu.

Jeśli spróbujemy otworzyć plik, który nie istnieje, otrzymamy komunikat o błędzie, a program zostanie przerwany. Możemy temu zaradzić, sprawdzając przed próbą otwarcia pliku, czy plik w ogóle istnieje. Tym razem skorzystamy z innego sposobu dostępu do pliku. Najpierw stworzymy zmienną `f` i przypiszemy jej typ plikowy `FILE`. Następnie z wykorzystaniem instrukcji warunkowej `if` spróbujemy otworzyć plik do odczytu — jest to domyślny sposób otwarcia pliku dla funkcji `open()`. Jeśli uda się uzyskać dostęp do pliku, zostanie zrealizowany blok kodu po instrukcji `if`. W przeciwnym razie blok kodu po instrukcji `else`. Do odczytu całej zawartości pliku tekstowego wykorzystamy funkcję `readAll()`:

```
var f: FILE

if open(f, "plik.txt"):
  echo "Hura! Plik istnieje!"
  echo f.readAll
  close(f)
else:
  echo "Plik nie istnieje..."
```

```
Hura! Plik istnieje!
To jest pierwszy wiersz.
To jest drugi wiersz.
Koniec.
```

Możemy także pobierać poszczególne linijki tekstu z pliku zamiast od razu całą jego zawartość. Linijki tekstu są odczytywane, licząc od pierwszej, funkcją `readLine()`. Każde jej wywołanie powoduje odczytanie kolejne linijki z pliku. Odczytajmy pierwsze dwie linijki tekstu:

```
var f: FILE

if open(f, "plik.txt"):
  echo "Hura! Plik istnieje!"
  echo f.readLine
  echo f.readLine
  close(f)
else:
  echo "Plik nie istnieje..."
```

```
Hura! Plik istnieje!
Pierwsza linijka.
Druga linijka.
```

Otwierając plik funkcją `open()`, musimy pamiętać, żeby taki plik, po zakończonych operacjach na nim, zamknąć funkcją `close()`. Dzięki temu uwalniamy dostęp do tego pliku.

Możemy wczytać całą zawartość pliku, a potem podzielić ten tekst na poszczególne linijki. Do tego celu wykorzystamy funkcję `splitLines()` z modułu `strutils`:

```
import strutils

let
  text = readFile("plik.txt")
  lines = text.splitLines
  echo lines
```

```
@["Pierwsza linijka.", "Druga linijka.", "Koniec."]
```

W wyniku takiego podziału otrzymaliśmy sekwencję, której zawartością są poszczególne linijki tekstu.

17.2 Zapis danych do pliku

Żeby zapisać dane do pliku tekstowego, możemy użyć funkcji `writeFile()`, podając argumenty: nazwa pliku (lub pełna ścieżka do niego) do zapisu i tekst, który chcemy

zapisać. Funkcja ta tworzy nowy plik, jeśli nie istnieje lub nadpisuje istniejący. Po zapisaniu danych do pliku zamyka go.

```
writeFile("plik.txt", "Hello world!")
```

Wróćmy jeszcze do funkcji `open()`, która dostarcza więcej opcji dostępu do pliku. Otworzymy sobie plik do zapisu, analogicznie, jak to zrobiliśmy w przykładzie powyżej:

```
var f = open("plik.txt", fmWrite)
f.write("Hello world!")

f.close
```

Poniżej znajduje się tabela z trybami otwierania plików (file mode, fm):

Tryb otwarcia pliku	**Opis**
`fmRead`	Tylko do odczytu.
`fmWrite`	Tylko do zapisu.
`fmReadWrite`	Do odczytu i zapisu. Jeśli plik nie istnieje, zostanie stworzony. Istniejący plik zostanie nadpisany.
`fmReadWriteExisting`	Do odczytu i zapisu. Jeśli plik nie istnieje, nie zostanie stworzony. Istniejący plik nie zostanie nadpisany.
`fmAppend`	Tylko do zapisu. Dane zostaną dołączone na końcu pliku.

Zatem, żeby dopisać nowy tekst do zawartości istniejącego pliku wykorzystamy `fmAppend`:

```
var f = open("plik.txt", fmAppend)
f.write("\nAla ma kota")

f.close
```