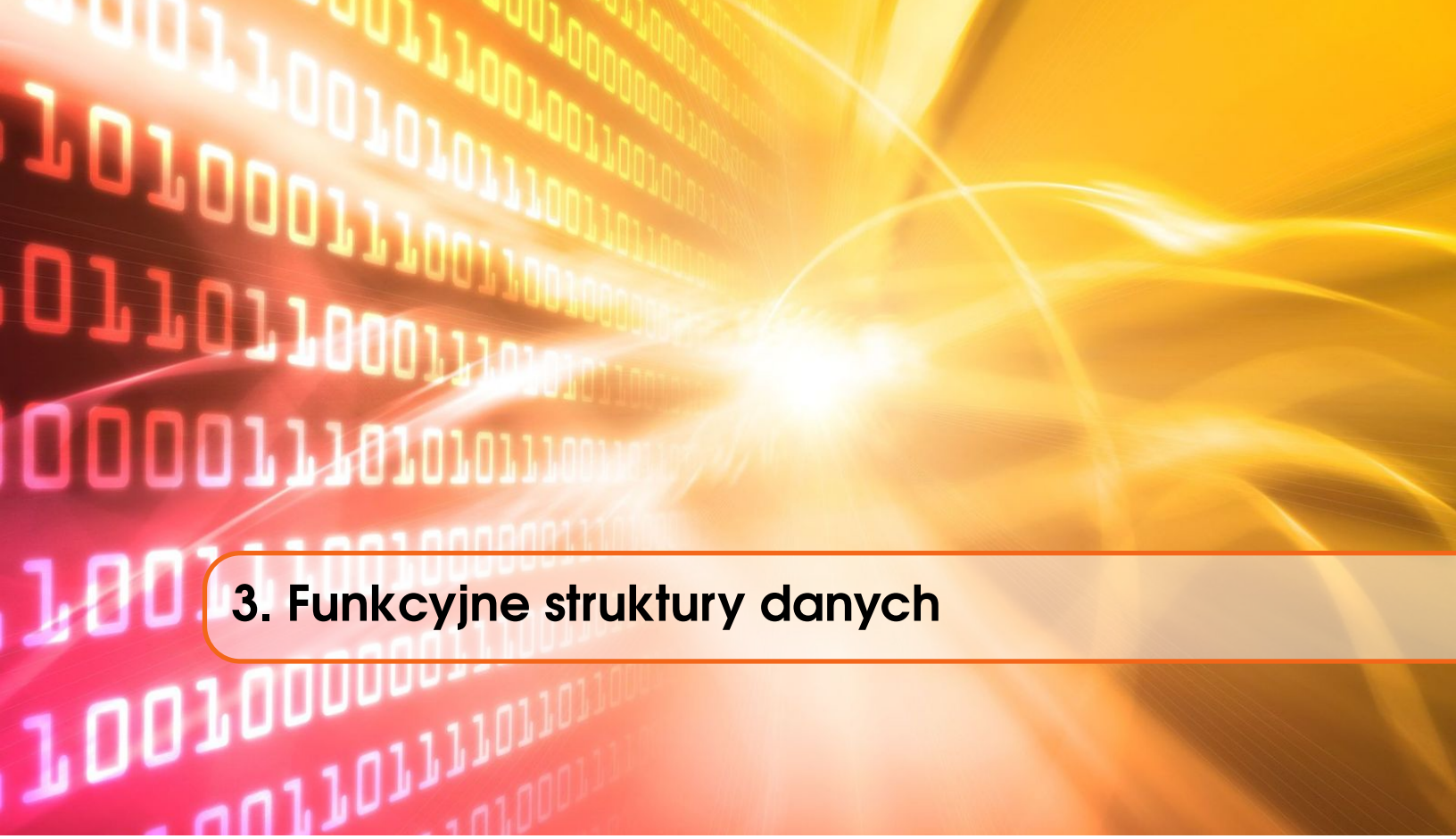
Programowanie funkcyjne

Materiały z wykładu i laboratorium

Łukasz Bartczuk



3. Funkcyjne struktury danych

Na tych zajęciach poznamy podstawowe funkcyjne struktury danych oraz podstawowe działania jakie można na nich wykonywać.

Oczywiście, w programach pisanych w F# możemy korzystać z dowolnych struktur danych dostępnych na platformie .NET. Są to listy, kolejki, słowniki itd. Należy jednak pamiętać, że większość z nich jest zaimplementowana w taki sposób aby jak najefektywniej współpracować z językami imperatywnymi takimi jak C# lub Visual Basic.

Ten sposób implementacji powoduje jednak, że nie współpracują one zbyt dobrze z językami funkcyjnymi takimi jak F# . Jest to szczególnie widoczne, gdy chcemy zachować podstawową cechę tych języków – niemodyfikowalność. Skoro wszystkie wartości proste lub złożone, są niemodyfikowalne w językach funkcyjnych, również tworzone struktury danych powinny takie być. Oznacza, to że wszystkie operacje zmiany elementów struktury (dodawanie nowego elementu, czy usuwanie go[1]) powinny utworzyć nową strukturę, bez wpływu na już istniejącą. Przykładowo, gdybyśmy chcieli stworzyć listę, sygnatury funkcji dodających nowy element na początek listy oraz usuwającej istniejący element, przyjęłyby następującą formę:

$$dodaj : Lista \rightarrow Element \rightarrow Lista$$

oraz

$$usun : Lista \rightarrow Element \rightarrow Lista$$

[1]Nie wspominam tutaj o zamianie elementu na inny, ze względu na to, że operację tę można zawsze przedstawić jako usunięcie danego elementu oraz dodanie nowego

Czyli obie te funkcje przyjmują jako swoje parametry listę wejściową oraz element, który chcemy do tej listy dodać lub z niej usunąć. Jako rezultat swojego działania funkcje te powinny zwracać nową listę z dodanym, lub usuniętym elementem. Niemutowalność tej struktury powoduje, że po uruchomieniu tych funkcji mamy dostęp zarówno do wersji listy sprzed ich wywołania, jak również do zmodyfikowanej, utworzonej w tych funkcjach listy.

W języku C# listy najczęściej tworzy się z wykorzystaniem bardzo popularnej klasy `List<>`. Jednak zastosowanie jej w programach funkcyjnych może znacząco obniżyć ich wydajność. Wynika to ze sposobu jej implementacji. Wewnętrznie klasa ta przechowuje dane w tablicy. Powoduje to, że podczas wywołania funkcji dodawania i usuwania elementu, aby zapewnić niemutowalność, konieczne byłoby utworzenie w pamięci nowej tablicy oraz skopiowanie elementów ze starej tablicy do nowej wraz z wykonaniem odpowiedniej operacji. Oczywiście zajmuje to czas i pamięć. Z tego powodu języki funkcyjne wykorzystują inny sposób implementacji tych struktur.

3.1 Listy

Matematycznie listę możemy przedstawić za pomocą następującego równania rekurencyjnego:

$$\text{Lista} = \begin{cases} () \\ \text{Element} :: \text{Lista} \end{cases}$$

Wynika z niego, że lista może być albo listą pustą (nie posiadającą żadnego elementu), oznaczaną jako $()$, albo jako para: Element :: Lista, gdzie Element jest pierwszym elementem listy, a Lista pozostałymi elementami na tej liście. Pierwszy element listy określa się mianem **głowy listy**, a pozostałe elementy na liście jej **ogonem**. Zastosowany tutaj operator :: nazywa się często konstruktorem listy, a funkcję go realizującą nazywa się **cons**. Funkcja ta służy nam również do tworzenia listy.

W praktyce najłatwiej takie listy zaimplementować jako tzw. **listy łączone**. Schematycznie lista taka przedstawiona jest na rysunku poniżej:

Rysunek 3.1: Budowa listy łączonej

Każdy węzeł takiej listy składa się z dwóch części: przechowywanych na niej danych oraz wskaźnika na następny jej element (w przypadku pierwszego węzła, części te to nic innego jak głowa listy i wskaźnik na jej ogon).

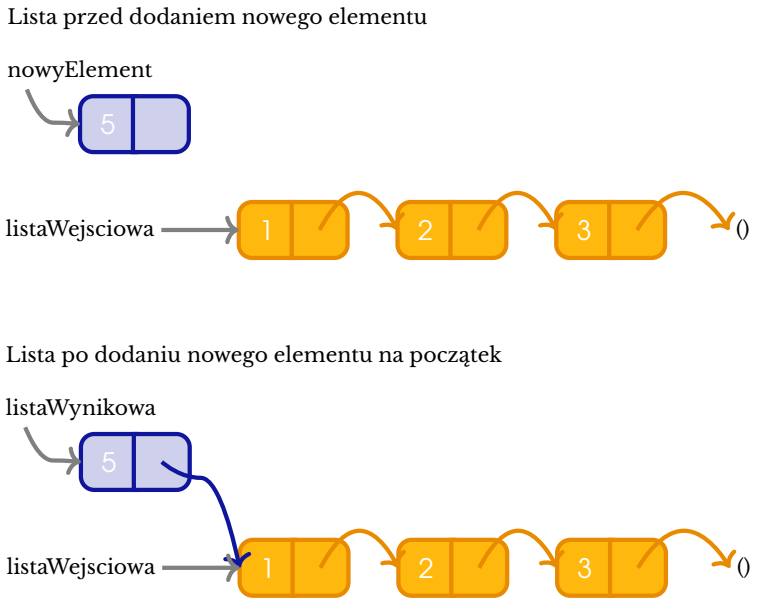
W jaki sposób implementacja ta może nam pomóc w zapewnieniu, że struktura będzie zarówno niemutowalna, ale również wydajna?

Przeanalizujmy tutaj operację dodawania nowego elementu na listę. Musimy rozpatrzeć dwa przypadki:

1. dodawanie elementu na początek listy łączonej
2. dodawanie elementu do środka listy łączonej

3.1.1 Dodawanie elementu na początek listy łączonej

Jest to najprostszy możliwy przypadek operacji na liście. Wystarczy dołączyć nowy element na początek listy. Załóżmy wobec tego, że istniejąca lista jest przypisana do zmiennej `listaWejsciowa`. Jeżeli dołączymy do niej nowy element, stanie się on jej nową głową. Jednak dopóki nie zmodyfikujemy zmiennej `listaWejsciowa`, to będzie ona dalej wskazywać na głowę oryginalnej listy, natomiast nowy wskaźnik, będzie pokazywał na głowę nowej.



Rysunek 3.2: Dodawanie nowego elementu na początek listy łączonej

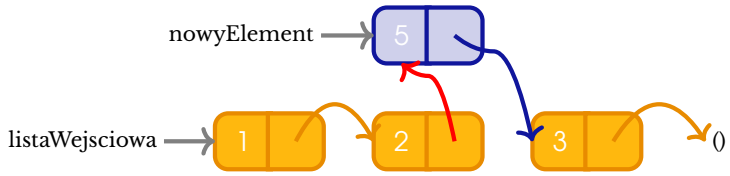
Jak widzimy dodawanie nowego elementu na początek listy łączonej nie powoduje żadnego dodatkowego obciążenia dla pamięci, ani potrzeby kopiowania elementów. Istniejącą listę możemy wykorzystać w całości, bez wpływania na jej kształt.

3.1.2 Dodawanie elementu do środka listy łączonej

Jest to zdecydowanie bardziej skomplikowana operacja. Załóżmy, że nowy element będziemy chcieli wstawić pomiędzy elementy "2"i "3". Jeżeli spróbujemy rozrysować tę operację na schemacie (patrz rysunek 2.3).

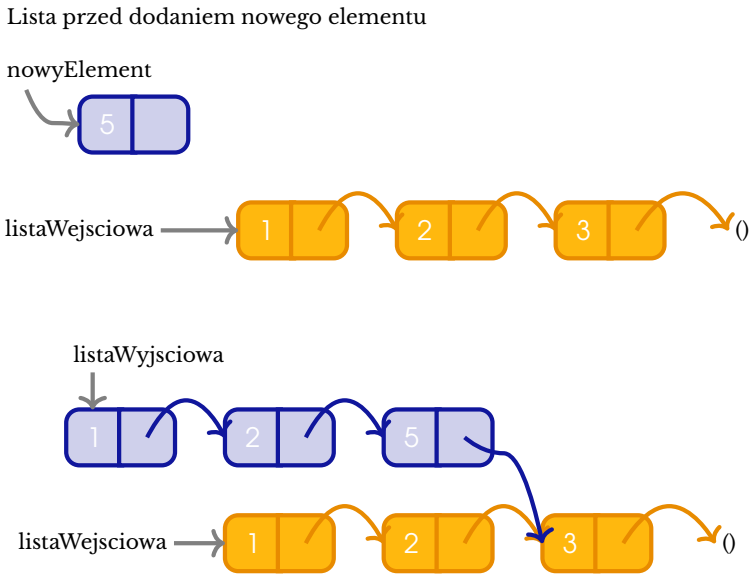
Z tego rysunku wynika, że bez problemu możemy ustawić wskaźnik nowego elementu tak, aby wskazywał na trzeci element. Jednak próba ustawienia wskaźnika drugiego elementu tab aby pokazywał na nowy spowoduje zmodyfikowanie listy wejściowej. Niestety nasza lista nie będzie dłużej niemodyfikowalna.

Zwróćmy jednak uwagę na fakt, że zaburzymy wyłącznie ten fragment listy, który



Rysunek 3.3: Dodawanie nowego elementu do środka listy łączonej

znajdzie się przed nowym elementem. Elementy za pozostaną nietknięte. Możemy ten fakt wykorzystać i ponownie użyć tych elementów. Dzięki temu, aby zachować niemodyfikowalność listy, konieczne jest skopiowanie tylko elementów, które w wynikowej liście będą znajdowały się przed nowo dodanym elementem. Jest to pokazane na rysunku 2.4.



Rysunek 3.4: Dodawanie nowego elementu do środka listy łączonej

Wynika z tego, że im bliżej początku listy chcemy dodać nowy element, tym nasza funkcja będzie wydajniejsza (więcej elementów można ponownie wykorzystać). Podobnie można zdefiniować operacje usuwania elementów.

3.2 Implementacja od podstaw listy łączonej w F#

Chodź oczywiście lista jest zdefiniowana w F# (będziemy ją analizować na następnych zajęciach), implementacja jej od podstaw, może być ciekawym ćwiczeniem.

Ponieważ jak pokazaliśmy wcześniej, podczas analizy listy należy rozważyć dwa osobne przypadki: listy pustej oraz listy zawierającej głowę i ogon. Wobec tego najlepszym typem do jej zaimplementowania będzie unia z dyskryminatorem:

```
type Lista<'a> =
| Pusta
| Wezel of 'a*Lista<'a>
```

W powyższym zapisie wykorzystałem możliwość utworzenia generycznej unii z dyskryminatorem. Umożliwia to uniezależnienie tworzonej struktury od konkretnego typu danych elementów, które będą na niej przechowywane.

Zastosowanie unii umożliwia utworzenie listy z rysunku 3.13 w następujący sposób:

```
    let lista = Wezel(1, Wezel(2, Wezel(3, Wezel (4, Pusta))))
```

Stwórzmy dodatkowo dwie funkcje, które mogą być przydatne podczas wykonywania operacji na listach `glowa` i `ogon`. Pierwsza zwróci początkowy element listy, a druga wszystkie elementy, poza początkowym.

```
let glowa =
  function
  | Pusta -> failwith "Nie mozna pobrac glowy z listy pustej"
  | Wezel(glowa,_) -> glowa

let ogon =
  function
  | Pusta -> failwith "Nie mozna pobrac ogona z listy pustej"
  | Wezel(_,ogon) -> ogon
```

Ponieważ lista jest strukturą rekurencyjną, to wszystkie operacje jakie będziemy na niej implementowali będą tworzone jako funkcje rekurencyjne. Przykładowo funkcja obliczająca liczbę elementów na liście może być zapisana w następujący sposób:

```
let rec liczbaElementow =
  function
  | Pusta -> 0
  | Wezel (_,ogon) -> licz ogon + 1
```

Myślę, że powyższa funkcja jest bardzo prosta i nie wymaga żadnego komentarza.

Za pomocą konstruktorów unii możemy utworzyć listę pustą lub dołączyć nowy element na początek listy. Jak wspomniano wcześniej jest to bardzo prosta operacja. Sprawa się jednak komplikuje, gdy będziemy chcieli dołączyć nowy element pomiędzy już istniejące elementy.

Przykładowo poniższa funkcja wyszukuje element na liście i nowy wstawia po jego pierwszym wystąpieniu. Musimy pamiętać jednak, że nie modyfikujemy istniejącej listy, tylko tworzymy nową ponownie wykorzystując jak najwięcej elementów:

```
let rec dodajPo element nowyElement =
  function
  | Pusta -> failwith ("Nie znalazlem elementu "
     + element.ToString())
  | Wezel (glowa,ogon) ->
      if glowa = element then
        Wezel(element, Wezel (nowyElement, ogon))
      else
        Wezel (glowa, dodajPo element nowyElement ogon)
```

Funkcja ta przyjmuje trzy parametry. `element` jest to poszukiwany element listy, po którym `nowyElement` ma być wstawiony. Ostatnim (ukrytym) parametrem funkcji jest lista, którą chcemy zmodyfikować.

Jeżeli lista, która jest przekazana jako ostatni (ukryty) parametr funkcji jest pusta, to oznacza, że nie może zawierać poszukiwanego elementu więc w tym przypadku rzucamy wyjątek. Jeżeli jednak istniejąca lista ma głowę i ogon, to gdy głowa jest równa poszukiwanemu elementowi tworzymy nową listę o następującej budowie: (szukanyElement::nowyElement::ogon istniejącej listy). W przypadku gdy głowa listy wejściowej nie jest poszukiwanym elementem to tworzymy nową listę, której głowa staje się pierwszym elementem istniejącej listy, a ogonem jest lista powstała poprzez wstawienie nowego elementu do ogona listy.

Wywołanie tej funkcji jest następujące:

```
dodajPo 2 4 lista
```

Powyższe przykłady, są zwykłymi funkcjami rekurencyjnymi. Spróbujmy je teraz zapisać w formie rekurencji ogonowej.

W przypadku funkcji `liczbaElementow` implementacja taka stosunkowo prosta i sprowadzi się do następującego kodu:

```
let liczbaElementow lista =
  let rec liczbaElementow suma =
    function
    | Pusta -> suma
    | Wezel(glowa, ogon) -> liczbaElementow (suma+1) ogon
  liczbaElementow 0 lista
```

Implementacja ta jest bardzo podobna do implementacji funkcji `silnia` pokazywanej wcześniej.

Dużo bardziej skomplikowane jest przekształcenie funkcji `dodajPo`. Przede wszystkim należy się zastanowić jakie trzy operacje wchodzą w skład tej funkcji. Są to:

1. Odnalezienie ogona listy, której głową jest element za którym chcemy wstawić nową wartość. Ogon ten będzie mógł być współdzielony pomiędzy obie listy (starą i nową)
2. Skopiowanie wszystkich elementów od początku listy do elementu, za którym chcemy wstawić nową wartość.
3. Połączenie list określonych w poprzednich punktach ze wstawieniem nowego elementu pomiędzy nie.

Każdy z tych punktów może być zaimplementowany jako osobna funkcja. Odpowiednio `ogonPoElemencie`, `kopiujDoElementu` i `polaczListy`:

```
let rec ogonPoElemencie element =
  function
  | Pusta -> failwith ($"Nie znalazlem elementu {element}")
  | Wezel (glowa,ogon) ->
      if glowa = element then
        ogon
      else
        ogonPoElemencie element ogon
```

```
let kopiujDoElementu element lista =
  let rec kopiujDoElementu element nowaLista =
    function
    | Pusta -> nowaLista
    | Wezel (glowa, ogon) ->
        let nowyElement = Wezel(glowa, nowaLista)
        if glowa = element then
          nowyElement
        else
          kopiujDoElementu element nowyElement ogon
  kopiujDoElementu element Pusta lista
```

```
let polaczListy lista1 lista2 =
  let rec polaczListy listaWynikowa =
    function
    | Pusta -> listaWynikowa
    | Wezel(glowa, ogon) ->
        polaczListy (Wezel (glowa, listaWynikowa)) ogon
  polaczListy lista2 lista1
```

Zwróćmy uwagę na sposób działania funkcji `kopiujDoElementu`. Wynikowa lista będzie odwrócona w stosunku do listy wejściowej, co pokazuje poniższe wywołanie:

```
let lista = Wezel(1, Wezel (2, Wezel (3, Wezel(4, Pusta))))
kopiujDoElementu 3 lista
```

W rezultacie uzyskamy następującą strukturę:

```
val it : Lista<int> = Wezel (3, Wezel (2, Wezel (1, Pusta)))
```

Zachowanie to jest związane z tym, że budując listę, nowy element dodajemy zawsze na jej początek. Wobec tego przechodząc po liście od pierwszego elementu z jednoczesnym tworzeniem nowej struktury nowa struktura musi być odwrócona.

Podobnie będzie działała funkcja `polaczListy`, która klonuje wszystkie elementy pierwszej listy w celu zapewnienia niezmienności obu list.

Wspomniane odwrócenie listy, wykonywane przez funkcję `kopiujDoElementu`, nie będzie dla nas jednak problemem, ponieważ funkcja `polaczListy` ustawi te elementy w odpowiedniej kolejności.

Ostatecznie cała procedura dodawania nowego elementu do środka listy, będzie przedstawiać się następująco:

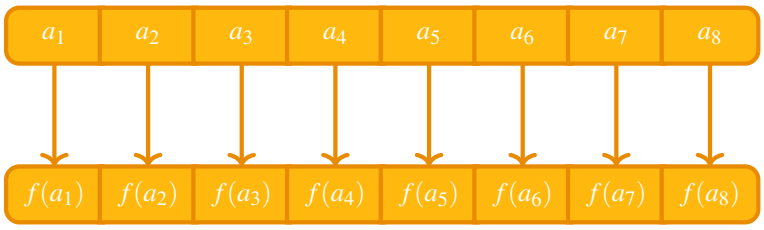
```
let dodajPo element nowyElement lista =
  let ogon = zwrocListePoElemencie element lista
  let poczatek = kopiujDoElementu element lista
  polaczListy  poczatek (Wezel (nowyElement, ogon))
```

3.3 Podstawowe operacje jakie możemy wykonywać na listach

W tym punkcie opisane zostaną podstawowe rodzaje operacji, jakie można wykonywać na listach.

3.3.1 Mapowanie

Mapowanie polega na stworzeniu nowej listy, o takiej samej liczbie elementów. Elementy na liście wynikowej mogą być takie same jak na liście wejściowej, lub podlegać pewnemu przekształceniu.



Rysunek 3.5: Operacja mapowania

Przykładem tej operacji może być stworzenie na podstawie istniejącej listy nowej , w której każdy element będzie dwukrotnie większy niż na liście wejściowej

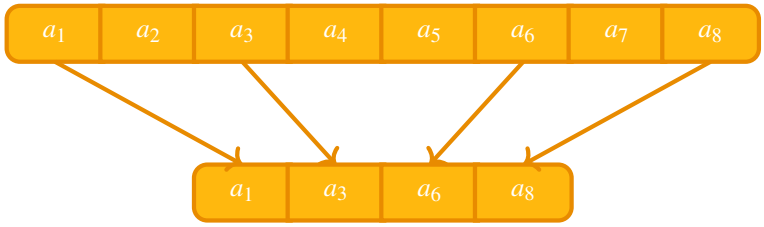
```
let rec dwaRazyWieksza = function
| Pusta -> Pusta
| Wezel (glowa, ogon) -> Wezel (glowa*2, dwaRazyWieksza ogon)
```

Funkcja ta pokazuje typowy szablon tego typu operacji. Jeżeli lista wejściowa jest listą pustą, to w rezultacie również można otrzymać tylko listę pustą. Jeżeli jednak lista wejściowa będzie się składała z głowy i ogona to:

1. głową nowej listy staje się wynik zdefiniowanej operacji zaaplikowanej do głowy listy wejściowej.
2. ogonem nowej listy jest rekurencyjne zaaplikowanie funkcji do ogona listy wejściowej

3.3.2 Filtrowanie

Filtrowanie polega na stworzeniu nowej listy o takiej samej lub mniejszej liczbie elementów. Ma liście wynikowej znajdują się tylko elementy, które spełniają określony warunek.



Rysunek 3.6: Operacja filtrowania

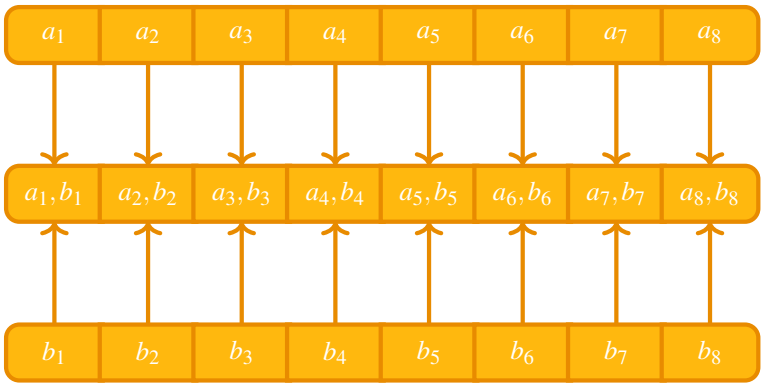
Przykładem tej operacji może być wybranie z listy wejściowej tylko elementów parzystych:

```
let rec tylkoParzyste = function
| Pusta -> Pusta
| Wezel (glowa,ogon) ->
    if glowa%2=0 then
      Wezel (glowa, tylkoParzyste ogon)
    else
      tylkoParzyste ogon
```

Szablon tej operacji jest podobny do mapowania, lecz teraz głową nowej listy staje się niezmodyfikowany, pierwszy element listy wejściowej lecz tylko wtedy, gdy spełnia on określony operacją filtrowania warunek

3.3.3 Zip

Operacja zip (połączenie) przyjmuje dwie listy jako parametry i tworzy nową listę, której elementami są pary elementów z list wejściowych.



Rysunek 3.7: Operacja zipowania

Przykładowa implementacja tej operacji pokazana jest poniżej:

```
let rec zip lista1 lista2 =
   match (lista1, lista2) with
   | (Pusta, Pusta) -> Pusta
   | (Wezel (glowa1,ogon1), Wezel (glowa2,ogon2))
     -> Wezel ((glowa1,glowa2), zip ogon1 ogon2)
   | (Pusta, _) | (_, Pusta)
     -> failwith "Obie listy powinny miec taka sama liczbe ↩
        elementow"
```

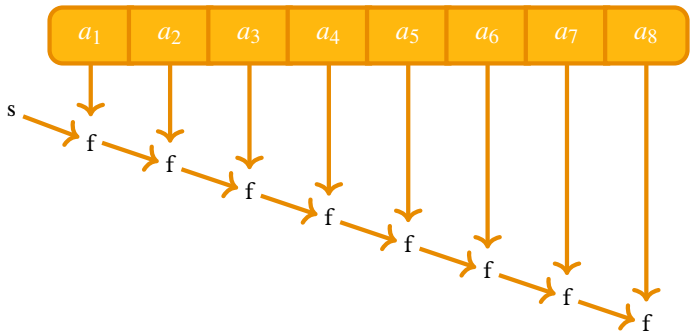
Ponieważ operacja ta wymaga dwóch list, jest ona bardziej skomplikowana niż poprzednie. Jeżeli obie listy wejściowe są puste, to lista wynikowa również będzie pusta. Jeżeli obie listy składają się z głowy i ogona to pierwszym elementem listy wynikowej będzie para głów obu list, a pozostałe jej element będą się składać z rezultatu operacji zipowania ogonów list wejściowych. Jeżeli jednak jedna z list będzie pusta, a druga nie, to funkcja zgłasza błąd ponieważ w tej podstawowej implementacji obie powinny mieć tę samą długość.

3.3.4 Agregacja

Agregacja polega na złączeniu listy w pojedynczą wartość.

Typowym przykładem operacji agregacji jest sumowanie elementów listy wejściowej. W tym przypadku element s z rysunku jest równy 0, a funkcją agregującą operator dodawania +.

```
let rec sumuj = function
```



Rysunek 3.8: Operacja agregacji na przykładzie sumowania elementów na liście

```
| Pusta -> 0
| Wezel(glowa, ogon) -> glowa + sumuj ogon
```

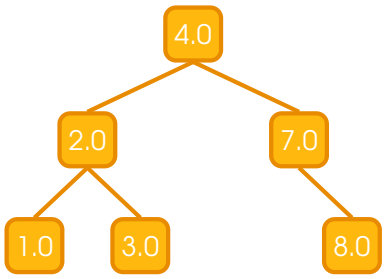
W przykładzie tym do kolejnych wywołań funkcji `f` przekazywaliśmy po prostu liczbę, nic nie stoi na przeszkodzie, aby przekazywać bardziej rozbudowane wartości np. rekordy. Odrobinę modyfikując wcześniejszy przykład możemy łatwo napisać funkcję obliczającą średnią elementów:

```
let srednia kolekcja =
  let rec srednia = function
  | Pusta -> {| sumaElementow = 0; liczbaElementow = 0 |}
  | Wezel(glowa, ogon) ->
      let nowyStan = srednia ogon
      {|
        sumaElementow = nowyStan.sumaElementow + glowa;
        liczbaElementow = nowyStan.liczbaElementow + 1
      |}
  let stan = srednia kolekcja
  stan.sumaElementow / stan.liczbaElementow
```

3.4 Drzewa

Listy są rekurencyjną strukturą danych świetnie pasującą do programowania funkcyjnego. Inną tego typu stukturą są drzewa. W tym punkcie rozpatrzymy szczególny przypadek takich struktur czyli uporządkowane drzewa binarne, ale na tej podstawie będziecie mogli przygotować aplikacje wykorzystujące również bardziej ogólne rodzaje drzew.

Drzewa binarne są to drzewa, w których każdy węzeł zawiera co najwyżej dwa drzewa potomne. Przykładowe tego typu drzewo przedstawione jest na rysunku 3.9.



Rysunek 3.9: Przykładowe uporządkowane drzewo binarne

Formalnie takie drzewo można zdefiniować za pomocą następującej formuły:

$$\text{Drzewo} = \begin{cases} () \\ \text{Wezel}(\text{wartość}, \text{Drzewo}, \text{Drzewo}) \end{cases} \tag{3.1}$$

Czyli drzewo może być drzewem pustym lub węzłem zawierającym wartość oraz dwa drzewa potomne (określane odpowiednio lewym i prawym poddrzewem). Element znajdujący się na szczycie drzewa określany jest jego korzeniem.

Drzewo uporządkowane jest to szczególny rodzaj drzewa binarnego, w którym wartości zawarte w lewym poddrzewie są mniejsze niż wartość umieszczona w korzeniu, a w prawym poddrzewie większe lub równe wartości w korzeniu. Dlatego na takim drzewie mogą być umieszczone tylko wartości, które można porównywać lub uporządkować.

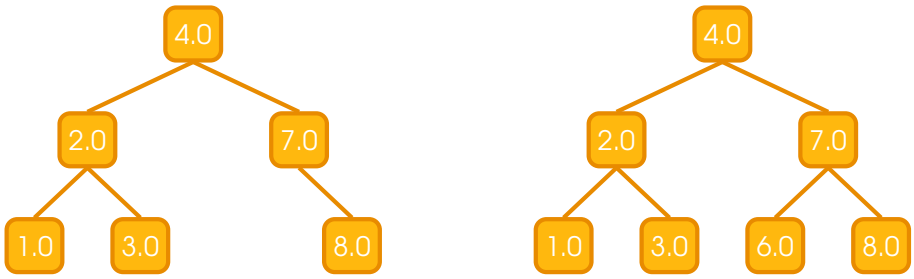
Poniżej przedstawiłem implementację typu danych opisującego uporządkowane drzewo binarne przechowujące wartości typu `float`.

```
type Drzewo =
  | Puste
  | Wezel of float * Drzewo * Drzewo
```

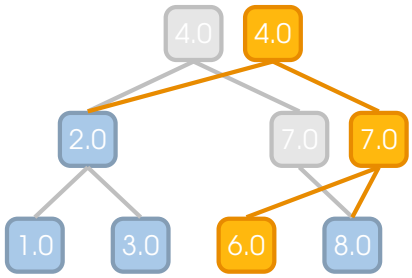
Oczywiście wszystkie operacje wykonywane na drzewie będziemy chcieli zaimplementować w taki sposób, aby z jednej strony zapewniały one niemodyfikowalność struktury, a z drugiej były efektywne. Dlatego będziemy chcieli jak najbardziej wykorzystać istniejącą strukturę, tak aby za każdym razem nie budować całego drzewa od początku.

Przykładowo wstawienie wartości 6.0 do drzewa przedstawionego na rys. 3.9, logicznie da nam następującą strukturę: Zwróćmy uwagę, że węzły, które będą wymagały zmiany zawierają wartości: 4.0, 7.0, czyli te, które znajdują się na ścieżce od korzenia drzewa do miejsca wstawienia tego węzła. Pozostałe cztery węzły mogą być ponownie wykorzystane. Graficzna reprezentacja drzew po dodaniu nowego węzła przedstawiona jest na rys. 3.11.

Implementacja funkcji realizującej tę oprację przedstawiona jest poniżej:



Rysunek 3.10: Operacja dodawania nowego węzła do binarnego drzewa uporządkowanego



Rysunek 3.11: Graficzna reprezentacja współdzielenia węzłów po operacji dodawania nowego elementu do binarnego drzewa uporządkowanego. Szarym kolorem oznaczone są węzły, które należy skopiować, niebieskim - te które są współdzielone pomiędzy nowym i starym drzewem, a pomarańczowe to węzły nowo utworzone

```
let rec wstaw liczba = function
  | Puste -> Wezel(liczba, Puste, Puste)
  | Wezel(x,l,p) when x<=liczba -> Wezel(x, l, wstaw liczba p)
  | Wezel(x,l,p) -> Wezel(x, wstaw liczba l, p)
```

Funkcja ta jest zaimplementowana jako zwykła operacja rekurencyjna więc jest podatna na wszystkie problemy jakie wiążą się z tego typu funkcjami. Zwróćmy uwagę, operacja ta może być rozbita na dwie osobne części. Pierwsza buduje ścieżkę (listę elementów prowadzącą od korzenia drzewa do miejsca wstawienia elementu), a druga tworzy na jej podstawie nowe drzewo. Obie te operacje bez większych problemów zapiszemy w formie rekurencji ogonowej:

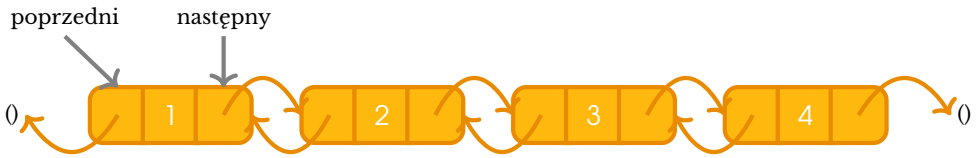
```
let wstaw liczba drzewo =

  let rec znajdzSciezke liczba sciezka = function
    | Puste -> Lista.Wezel(Puste, sciezka)
    | Wezel (x,_,p) as w when x<=liczba ->
        znajdzSciezke liczba (Wezel (w,sciezka)) p
```

```
    | Wezel (_,l,_) as w ->
        znajdzSciezke liczba (Wezel (w,sciezka)) l

let rec zbuduj liczba drzewo = function
  | Pusta -> drzewo
  | Lista.Wezel (w,ogon) ->
      let nowyWezel =
        match w with
        | Puste -> Wezel(liczba, Puste, Puste)
        | Wezel(x,l,_) when x<=liczba-> Wezel(x, l, drzewo)
        | Wezel(x,_,p) -> Wezel(x, drzewo, p)
      zbuduj liczba nowyWezel ogon

let sciezka = znajdzSciezke liczba Pusta drzewo
zbuduj liczba Puste sciezka
```

3.5 Zipper

Przedstawione w poprzednich sekcjach lista i drzewo są strukturami jednokierunkowymi. Po liście możemy się poruszać tylko od jej początku w kierunku końca, a po drzewie tylko w głąb. Powrót do wcześniejszego elementu może być, jak widzieliśmy, operacją czasochłonną.

W językach imperatywnych efektywna implementacja tej operacji dla list wymaga zastosowania tzw. list dwukierunkowych, których budowa przedstawiona jest na poniższym rysunku[2]:

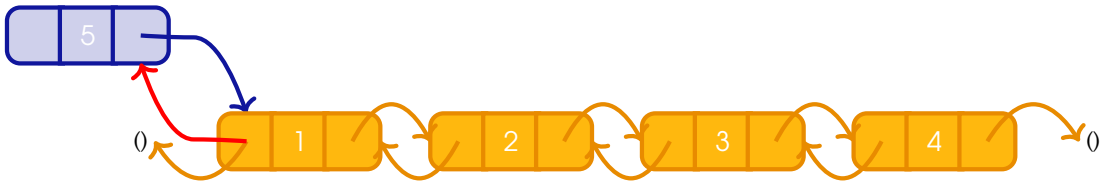


Rysunek 3.12: Budowa dwukierunkowej listy łączonej

Jak widzimy, każdy element listy zawiera dwa wskaźniki. "Następny"pokazuje na kolejny element na liście, a "poprzedni"na wcześniejszy. Dodawanie nowego elementu na początek listy wymaga zmodyfikowania wskaźnika poprzedni aktualnie pierwszego jej elementu, co pokazuje rysunek.

Takie działanie jednak powoduje, ze listy dwukierunkowej nie da się zaimplementować jako struktury niemodyfikowalnej.

[2]Podobnie można to zaimplementować w przypadku drzewa – każdy węzeł zawiera wskaźnik na węzeł nadrzędny



Rysunek 3.13: Budowa dwukierunkowej listy łączonej

Problem ten rozwiązany został przez Gerharda Huet'a, który w artykule Functional Pearl: The Zipper zaproponował nowy rodzaj struktury danych - zipper.

Opiera się ona na bardzo prostej idei – budowania listy odwiedzonych już podczas przechodzenia po liście elementów:

Dla listy łączonej w F# zipper może być przedstawiony w formie następującego rekordu:

Kod 3.1: Deklaracja typu Zipper dla listy łączonej

```
type Zipper<'a> = {
    Odwiedzone:Lista<Lista<'a>>;
    Obecny:Lista<'a>
}
```

Zawiera on dwie składowe. Pierwsza z nich – `Odwiedzone` – jest to lista odwiedzonych już węzłów listy, a druga `Obecny` wskazuje na aktualny element listy.

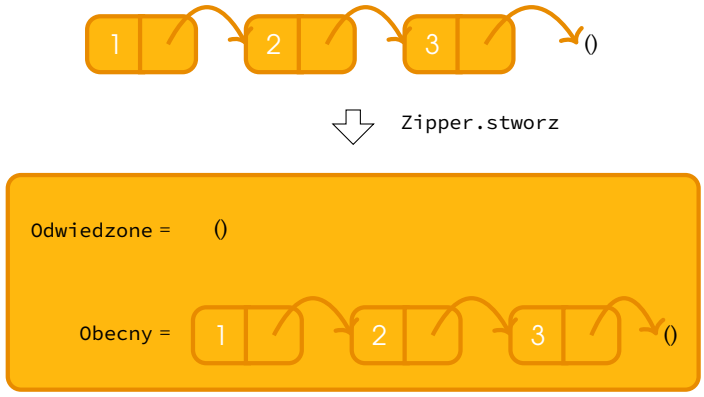
Możemy też stworzyć moduł zawierający funkcje do obsługi naszej kolekcji:

Kod 3.2: Deklaracja modułu Zipper

```
[<AutoOpen>]
module Zipper =
    let stworz lista = { Odwiedzone=Pusta; Obecny=lista }

    let nastepny {Odwiedzone = odwiedzone; Obecny = obecny}=
        match obecny with
        | Pusta -> failwith "To jest ostatni element listy"
        | Element (_,ogon) as el ->
            {Odwiedzone=el+odwiedzone; Obecny = ogon}

    let prior { Odwiedzone = odwiedzone; Obecny = _ } =
        match odwiedzone with
        | Pusta -> failwith "To jest pierwszy element listy"
        | Element (glowa, ogon) ->
            {Odwiedzone=ogon; Obecny = glowa}
```

```
    let aktualny zipper = Lista.glowa zipper.Obecny
```

Zdefiniowałem w nim cztery proste funkcje. Pierwsza `stworz`, która na podstawie listy tworzy nowy zipper. Składowa `Obecny` wskazuje na pierwszy element listy dla, której ten zipper jest utworzony, a składowa `Odwiedzone` jest listą pustą (w końcu nie przesunęliśmy jeszcze naszego zippera więc nie ma wcześniejszych elementów). Graficzna reprezentacja działania tej funkcji pokazana jest na rys. 3.14.



Rysunek 3.14: Graficzna reprezentacja działania funkcji `Zipper.stworz`

Druga funkcja modułu `Zipper` – `nastepny`, przesuwa nasz obiekt na następny element na liście. W tym celu, wartość składowej `Obecny` ustawiana jest na ogon listy, który jest do niej przypisany. Z kolei na początek listy zawartej w składowej `Odwiedzone` dodawana jest aktualna zawartość składowej `Obecny`. Ponieważ rekordy są strukturą tylko do odczytu funkcja ta nie modyfikuje istniejących danych tylko tworzy nowe. W ten sposób nasz zipper również jest kolekcją niemodyfikowalną. Graficzna reprezentacja dwukrotnego zastosowania funkcji `Zipper.nastepny` do struktury utworzonej na rys. 3.14 przedstawiona jest na rys. 3.15.

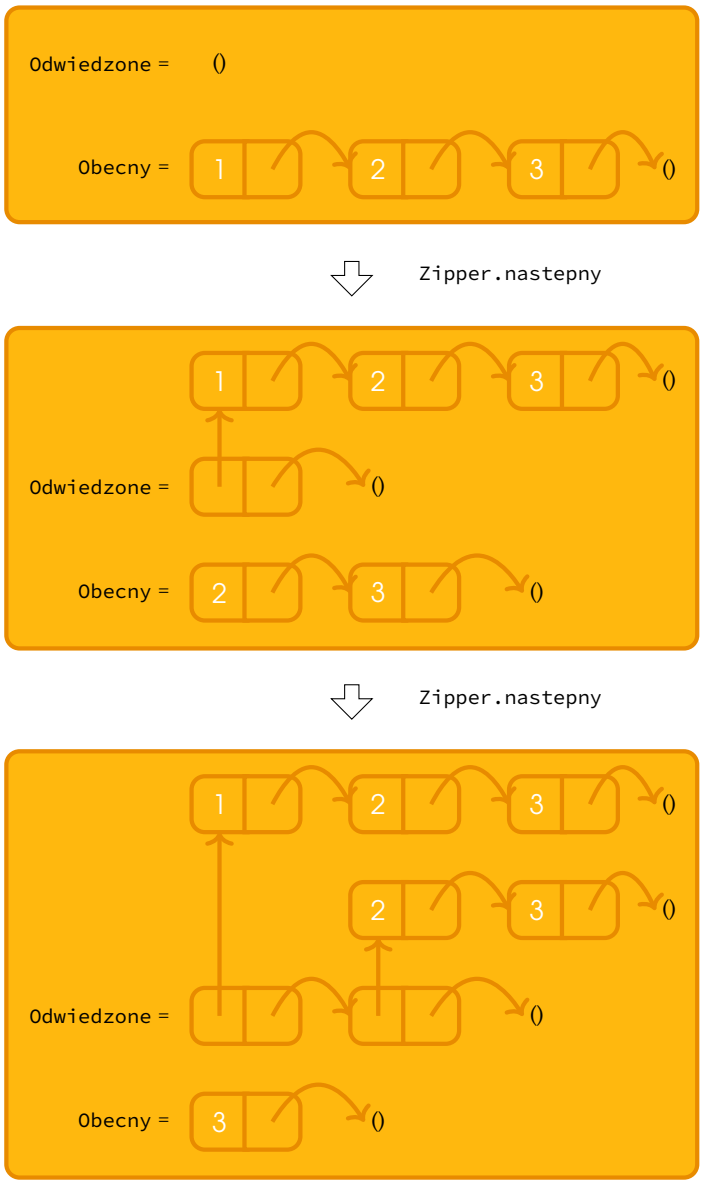
Funkcja `Zipper.poprzedni` działa odwrotnie w stosunku do przedstawionej wcześniej funkcji `Zipper.poprzedni` tzn. z kolekcji `Odwiedzone` usuwa pierwszy element i ustawia go jako wartość składowej `Obecny`. Oczywiście to również odbywa się w sposób nieniszczący.

Ostatnia z pokazanych funkcji `Zipper.aktualny` pobiera wartość z elementu listy, na który aktualnie wskazuje zipper. Przykład zastosowania przedstawionych funkcji pokazany jest w kodzie 3.3.

W podobny sposób możemy zdefiniować Zipper dla struktury drzewiastej. Zadanie to jednak pozostawiam jako ćwiczenie.

Kod 3.3: Przykład zastosowania funkcji modułu `Zipper`

```
let lista = Element(1, Element(2, Element(3, Pusta)))
let zipper = Zipper.stworz lista
printfn "%d" (Zipper.aktualny zipper)
```



Rysunek 3.15: Graficzna reprezentacja dwukrotnej aplikacji funkcji `Zipper.nastepny`

```
let zipper1 = Zipper.nastepny (Zipper.nastepny zipper)
printfn "%d" (Zipper.aktualny zipper1)
let zipper2 = Zipper.porzedni zipper2
printfn "%d" (Zipper.aktualny zipper1)
```

3.6 Zadania

Zadanie 3.1 Napisz funkcję, która generuje listę zbudowaną z n pierwszych liczb naturalnych. Sygnatura funkcji powinna przyjmować następującą postać:
`nPierwszych: n:int->Lista<int>`

Zadanie 3.2 Napisz funkcję, która generuje listę liczb całkowitych z określonego przedziału min, max. Funkcja powinna również pozwalać określić *krok* z jakim wartości będą generowane. Pierwszą wartością na liście powinna być min, a ostatnia powinna być mniejsza lub równa max

Zadanie 3.3 Napisz funkcję, która zwróci n-ty element listy.

Zadanie 3.4 Napisz funkcję, która określi czy dany element znajduje się na liście.

Zadanie 3.5 Napisz funkcję, która określi indeks podanego elementu. Jeżeli element nie znajduje się na liście zwróć odpowiednią wartość (możesz wykorzystać unie z dyskryminatorem).

Zadanie 3.6 Napisz funkcję, która usuwa z listy element na podanej pozycji.

Zadanie 3.7 Napisz funkcję pozwalającą obliczyć średnią wartości na liście.

Zadanie 3.8 Napisz funkcję, która pozwoli połączyć tablicę stringów w jeden łańcuch znaków. Funkcja powinna przyjmować parametr separator, który określa znak lub znaki jakimi należy rozdzielić poszczególne łańcuchy.

Zadanie 3.9 Napisz funkcję, która będzie przyjmowała listę stringów oraz zwracała listę liczb całkowitych zawierającą długości poszczególnych łańcuchów znaków

Zadanie 3.10 Napisz funkcję, która będzie przyjmowała listę stringów oraz wyszukiwała najdłuższy i najkrótszy wyraz

Zadanie 3.11 Napisz funkcję, która będzie przyjmowała listę stringów reprezentujących polskie imiona i zwróci tylko listę imion żeńskich.

Zadanie 3.12 Napisz funkcję, która będzie odwracała kolejność elementów na liście.

Zadanie 3.13 Napisz funkcję, która będzie przyjmowała listę stringów reprezentujących polskie imiona i zwróci dwie listy: osobno listę imion żeńskich oraz listę imion męskich.

Zadanie 3.14 Napisz funkcję, która będzie przyjmowała dwie listy liczb całkowitych i zwracała listę wartości logicznych, gdzie `true` określa, że liczba na pierwszej liście była większa, a `false`, że wartośc na drugiej liście była większa. Jeżeli jedna lista jest dłuższa od drugiej zwróć wyjątek informujący o tym fakcie.

Zadanie 3.15 Napisz funkcję, która będzie przyjmowała dwie listy liczb całkowitych i zwracała listę wartości zdefiniowanego przez ciebie typu wyliczeniowego, gdzie `Pierwsza` określa, że liczba na pierwszej liście była większa, a `Druga`, że wartośc na drugiej liście była większa. Jeżeli jedna lista się skończy, to generowanie listy wejściowej ma trwać dalej dopóki są elementy na drugiej liście. W tym przypadku przyjmujemy, że istniejący element jest większy niż nieistniejący.

Zadanie 3.16 Napisz funkcję, która sprawdzi, czy lista elementów jest posortowana. Kierunek sortowania (malejący lub rosnący) powinien być zdefiniowany jako typ wyliczeniowy i przekazywany do funkcji.

Zadanie 3.17 Napisz funkcję, która będzie przyjmowała dwie posortowane listy liczb całkowitych (listy powinny być posortowane w tym samym kierunku). Funkcja powinna łączyć te listy w jedną z zachowaniem porządku.

Zadanie 3.18 Zaimplementuj stos wykorzystując przedstawioną listę łączoną

Zadanie 3.19 Zaimplementuj mapę wykorzystując przedstawioną listę łączoną. Na liście przechowuj pary elementów (klucz; wartość). Dostęp do dowolnego elementu powinien

następować po podaniu klucza.

Zadanie 3.20 Napisz funkcję, która będzie zliczała liczbę elementów na drzewie binarnym.

Zadanie 3.21 Napisz funkcję, która oblicza sumę wartości przechowywanych w drzewie binarnym

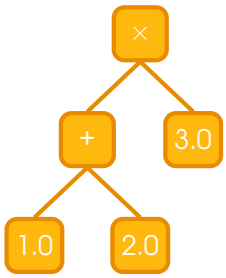
Zadanie 3.22 Napisz funkcję pozwalającą usunąć element z uporządkowanego drzewa binarnego.

Zadanie 3.23 Napisz funkcję pozwalającą określić czy w uporządkowanym drzewie binarnym znajduje się podana wartość.

Zadanie 3.24 Ścieżką do węzła n w drzewie nazywamy kolekcję wszystkich węzłów prowadzących od korzenia do tego węzła. Przykładowo, dla drzewa z rys. 3.9, ścieżka do węzła 8.0 to $\{4.0, 7.0, 8.0\}$. Napisz funkcję zwracającą ścieżkę do węzła z określoną wartością (dla uproszczenia zakładamy, że wartości w drzewie się nie powtarzają).

Zadanie 3.25 Napisz funkcję, która dla drzewa binarnego zwraca jego głębokość, czyli liczbę elementów w jego najdłuższej gałęzi.

Zadanie 3.26 Niech drzewo reprezentuje wyrażenia arytmetyczne np: Zdefiniuj odpo-



Rysunek 3.16: Drzewo reprezentujące wyrażenie $(1.0+2.0)*3.0$.

wiednie struktury do opisu takiego drzewa oraz napisz funkcje, która pozwoli wyznaczyć wartość takiego wyrażenia. Możliwe do zastosowania operatory to: '+', '-', '*', '/' ,'^' (potęgowanie)

Zadanie 3.27 Rozszerz poprzedni program tak, aby wspierał również funkcje np. sin, cos lub inne z typu `Math`

Zadanie 3.28 Rozszerz poprzednie programy tak, aby wspierały również zmienne (symbole), których wartość może być ustalona przed obliczeniem wartości wyrażenia.

Zadanie 3.29 Jedno z zadań przedstawionych na poprzednich zajęciach dotyczyło wprowadzania i wyświetlania informacji o osobie (zadanie 2.15). Rozszerz je, tak aby dało się wprowadzać dowolną liczbę osób. Rozszerz typ Osoba dodając do niego jednoznaczny identyfikator danej osoby (określ co to może być). Funkcje pokaż rekord i modyfikacja rekordu powinny pozwolić określić, na którym rekordzie użytkownik chce wykonać działania. Dodaj również funkcje pozwalające wyszukać wszystkie osoby o podanym nazwisku (i wyświetlić listę tych osób na ekranie) oraz usunąć wybraną osobę.

Zadanie 3.30 Napisz program, który będzie pozwalał użytkownikowi na wprowadzanie

z klawiatury liczb oraz operatorów i funkcji matematycznych w odwrotnej notacji polskiej (przy czym my zakładamy, że każdy element jest podawany w nowej linii i zatwierdzany osobno). Po podaniu symbolu '=' program powinien obliczać wartość podanego wyrażenia. Przykładowo:

```
10.0
20.0
+
sin
=
```

jest innym sposobem zapisania wyrażenia $\sin(10.0+20.0)$ i jego wykonanie powinno na ekranie wyświetlić: -0.9880316241. Jeżeli użytkownik poda za mało wartości do przeprowadzenia danej operacji na ekranie należy wyświetlić stosowny komunikat np.

```
10.0
+
=
```

powinno wyświetlić komunikat: "Operator + wymaga podania dwóch wartości".