

Wstęp do programowania (potok funkcyjny)

Marcin Kubica

2017/2018

Spis treści

Wstęp	5
Podstawy języka programowania	15
Dekompozycja problemu, weryfikacja rozwiązania	30
Struktury danych	38
Moduły	62
Procedury wyższych rzędów jako abstrakcje konstrukcji programistycznych	75
Procedury wyższych rzędów i listy	86
Model obliczeń	103
Analiza kosztów	115
Zasada „dziel i rządź” i analiza złożoności na przykładzie algorytmów sortowania	132
Zasada „dziel i rządź” i analiza złożoności na przykładzie algorytmów sortowania, c.d.	140
Funktory	152
Programowanie imperatywne	161
Imperatywne wskaźnikowe struktury danych	173
Logika Hoare’a — dowodzenie poprawności programów imperatywnych	190
Przeszukiwanie grafów	200
Back-tracking	213

Technika spamiętywania	225
Programowanie dynamiczne	229
Kody Huffmana i algorytmy zachłanne	241
Algorytmy zachłanne c.d.	249
Algorytm mini-max	256
Gry typu wygrana/przegrana i drzewa AND-OR	267
Wyszukiwanie wzorców	273
Tablica sufiksowa	279
Procedury dużo wyższych rzędów	281
Strumienie	290
Programowanie obiektowe	300
Co jeszcze?	304

Kilka słów na początek

0.1 Programowanie funkcyjne vs. imperatywne

Cel jaki chcemy osiągnąć za pomocą programu możemy opisać za pomocą pojęć z dziedziny algorytmicznej, jako funkcje/relacje między danymi, a wynikami. Tak więc nasz cel jest pewnym tworem matematycznym. Do osiągnięcia tego celu możemy podejść na dwa sposoby:

Imperatywnie: Dane są to pewne przedmioty, na których należy wykonywać określone czynności, w wyniku których przeistoczą się one w wyniki. Podejście to bierze się stąd, że zwykle dane i wyniki modelują elementy otaczającego nas świata, a ten składa się z przedmiotów i zachodzą w nim zdarzenia. W tym podejściu mówimy procesom, jakie czynności mają kolejno wykonywać.

Funkcyjnie: Zarówno podstawowe pojęcia i typy, jak i nasz cel są pojęciami matematycznymi. Nasze programy opisują, jak z prostszych pojęć matematycznych konstruować bardziej skomplikowane, aż uda nam się skonstruować matematyczny obiekt odpowiadający naszemu celowi. Ponieważ często konstruujemy tu różnego rodzaju funkcje — stąd nazwa.

Stosując podejście funkcyjne łatwiej uzyskać pewność, że osiągniemy zamierzony cel, choć nie zawsze łatwo uzyskać efektywny program.

Charakterystyczne dla programowania funkcyjnego jest również to, że funkcje są „obywatelami pierwszej kategorii”, tzn. przysługują im wszystkie te prawa, co innym wartościom. W tej chwili nie jest jeszcze jasne o jakie prawa może chodzić. Można jednak śmiało powiedzieć, że jest to jedna z fundamentalnych zasad programowania funkcyjnego.

0.2 Dwa potoki: imperatywny i funkcyjny

Dlaczego funkcyjnie? Nie chcemy nauczyć Państwa języka programowania, ale pewnych technik tworzenia programów. Stosując programowanie funkcyjne odrywamy niektórych z Państwa od już nabytych złych nawyków. Jest to nieustający eksperyment — coroczny.

Jeśli nie jesteś pewien swoich sił lub nie programowałeś wcześniej w żadnym języku imperatywnym, lepiej wybierz potok imperatywny. Jeśli znasz Pascala, C lub C++, to programowanie funkcyjne może być ciekawsze.

0.3 Kwestie organizacyjne

- Zasady zaliczenia — 2–3 kolokwia + laboratorium + ew. egzamin pisemny.
- Zadania z kolokwiów i egzaminów z lat poprzednich zostały umieszczone w niniejszym skrypcie, choć nie zostało to w żaden szczególny sposób zaznaczone.

0.4 Podziękowania

Niniejsze materiały powstały na podstawie notatek do prowadzonych przeze mnie, na przestrzeni kilku lat, wykładów ze Wstępu do programowania i Metod programowania (potok funkcyjny). Chciałbym gorąco podziękować moim kolegom, którzy w tym czasie prowadzili ćwiczenia z tych przedmiotów: Jackowi Chrzęszczowi, Bogusławowi Kluge, Mikołajowi Konarskiemu, Łukaszowi Lwowi, Robertowi Maronowi i Kubie Pawlewiczowi. Ich uwagi i propozycje

miały wpływ na postać prowadzonych zajęć, a więc i również na te materiały. W szczególności część zamieszczonych tu zadań pochodzi od nich.

Szczególnie chciałbym podziękować Piotrowi Chrzastowskiemu, który prowadził równoległe ze mną wykłady ze Wstępu do programowania i Metod programowania dla potoku imperatywnego. Współpracując ze sobą wymienialiśmy się różnymi pomysłami. Tak więc podobieństwo pewnych elementów materiałów do naszych wykładów jest nie tylko nieuniknione, ale wręcz zamierzone. W szczególności, niektóre zamieszczone tutaj zadania pochodzą z materiałów opracowanych przez Piotra i umieszczonych w portalu: <http://wazniak.mimuw.edu.pl>.

Podziękowania należą się również Krzysztofowi Diksowi, za to, że nakłonił mnie do przetłumaczenia na język polski książki H. Abelsona i G. J. Sussmana, *Struktura i interpretacja programów komputerowych*. Książka ta była punktem wyjścia przy opracowywaniu programu zajęć na potoku funkcyjnym, w pierwszych latach jego istnienia. Jej dogłębne poznanie dało mi dobre podstawy do poprowadzenia wykładów. Na koniec chciałbym podziękować mojej żonie Agnieszce, za to, że wytrzymała ze mną gdy tłumaczyłem wspomnianą książkę.

0.5 Literatura

- [HA02] H. Abelson, G. J. Sussman, *Struktura i interpretacja programów komputerowych*, WNT 2002.
- [Ler] X. Leroy, *The Objective Caml system*, <http://caml.inria.fr/pub/docs/manual-ocaml/index.html>.
- [Kub] M. Kubica, *Programowanie funkcyjne*, notatki do wykładu, http://wazniak.mimuw.edu.pl/index.php?title=Programowanie_funkcyjne.
- [CMP] E. Chailloux, P. Manoury, B. Pagano, *Developing Applications with Objective Caml*, <http://caml.inria.fr/pub/docs/oreilly-book/>.
- [Ben08] J. Bentley, *Perłki Oprogramowania*, WNT 2008.

Wykład 1. Wstęp

1.1 Programowanie jako dziedzina magii

Wbrew powszechnie panującym opiniom programowanie jest gałęzią magii. W „Nowej encyklopedii powszechnej PWN” możemy znaleźć następujące określenie magii: „zespół działań, zasadniczo pozaempirycznych, symbolicznych, których celem jest bezpośrednio osiągnięcie [...] pożądanych skutków [...]”. Wyróżniamy przy tym następujące składniki działań magicznych:

- zrytualizowane działania (manipulacje),
- materialne obiekty magiczne (amulety, eliksiry itp.),
- reguły obowiązujące przy praktykach magicznych (zasady czystości, reguły określające czas i miejsce rytuałów),
- formuły tekstowe mające moc sprawczą (zaklęcia).

Programowanie mieści się w ramach ostatniego z powyższych punktów. Programy komputerowe są zapisanymi w specjalnych językach (zwanymi językami programowania) zaklęciami. Zaklęcia te są przeznaczone dla specjalnego rodzaju duchów żyjących w komputerach, zwanych procesami obliczeniowymi. Ze względu na to, że komputery są obecnie produkowane seryjnie, stwierdzenie to może budzić kontrowersje. Zastanówmy się jednak chwilę, czym charakteryzują się duchy. Są to obiekty niematerialne, zdolne do działania. Procesy obliczeniowe ewidentnie spełniają te warunki: nie można ich dotknąć ani zobaczyć, nic nie ważą, a można obserwować skutki ich działania, czy wręcz uzyskać od nich interesujące nas informacje.

Nota bene, w trakcie zajęć można się spotkać również z przejawami pozostałych wymienionych składników działań magicznych. „Logowanie” się do sieci oraz wyłączanie komputera to działania o wyraźnie rytualnym charakterze. Przedmioty takie jak indeks, czy karta zaliczeniowa wydają się mieć isticie magiczną moc.

W trakcie zajęć z tego przedmiotu zajmiemy się podstawowymi zasadami formułowania zaklęć/programów, a także związkami łączącymi zaklinającego (programistę), program i proces obliczeniowy. W szczególności zajmiemy się również analizą sposobu, w jaki procesy obliczeniowe realizują programy oraz metodami upewniania się, że programy realizują zamierzone cele. Będziemy jednak pamiętać, że zaklęcia są przeznaczone dla procesów żyjących w komputerze, stąd będziemy je wyrażać w języku programowania *Ocaml*. Nie jest to jednak kurs programowania w tym języku, lecz kurs podstawowych zasad formułowania zaklęć dla procesów obliczeniowych.

1.2 Kilka podstawowych pojęć

Jako adepci magii związanej z zaklinaniem procesów obliczeniowych będziemy nazywać siebie **informatykami**. W naszych działaniach będziemy sobie stawiać **cele**, które chcemy osiągnąć i, używając rozumu, będziemy formułować zaklęcia mające zmusić procesy obliczeniowe do doprowadzenia do postawionych celów oraz upewniać się, że cele te będą osiągnięte. Formułowane zaklęcia będziemy nazywać **programami**, a ich formułowanie **programowaniem**. Upewnianie się, co do skutków zaklęć będziemy nazywać **weryfikacją** programów. **Procesy obliczeniowe** nie są zbyt inteligentnymi duchami, za to są bardzo szybkie. Dlatego też programy muszą szczegółowo opisywać czynności, jakie mają wykonać te duchy. Sposób

postępowania opisany przez program, w oderwaniu od sposobu jego zapisu w konkretnym języku programowania będziemy nazywać **algorytmem**. Inaczej mówiąc: *program = algorytm + sposób zapisu*.

1.3 Algorytm

Muhammad Ibn Mūsā al-Khwārizmi ok. 780–850 r. n.e. perski matematyk, geograf i astronom działający za czasów kalifa Al-Ma'mūna, w Bagdadzie w *Domu Mądrości* (bibliotece i ośrodku tłumaczeń założonym przez kalifa Haruna Al-Rashida). Autor:

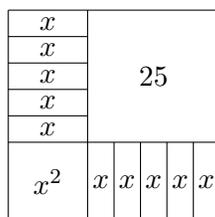
- *Tablic* — astronomia, kalendarz, funkcje trygonometryczne,
- *Arytmetyki* — hinduski zapis pozycyjny, metody rachunkowe,
- *Algebry (Zasad redukcji i przenoszenia, Kitāb al-jabr wa'l-muqābala*, od czego pochodzi słowo *algebra*) — równania liniowe i kwadratowe,
- *Kalendarza żydowskiego, Kronik, Geografii i Astrologii*.

Jego prace przyczyniły się do rozpowszechnienia cyfr arabskich i zapisu pozycyjnego. Od jego nazwiska pochodzi słowo **algorytm** (Al-Chuwarizmi, Alchwarizmi, Algorithmi, Algorithmus).

Oto przykład z *Algebry* dotyczący metody rozwiązywania równań kwadratowych. Liczby utożsamiamy tu z długościami odcinków, a mnożenie z powierzchnią prostokątów. W tych czasach pojęcie liczb ujemnych i zera nie było jeszcze znane.

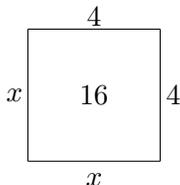
a	b	c	równanie	przykład	komentarz
+	+	+	$ax^2 + bx + c = 0$	—	—
+	+	0	$ax^2 + bx = 0$	—	—
+	+	—	$ax^2 + bx = c$	$x^2 + 10x = 39$	metoda 1
+	0	+	$ax^2 + c = 0$	—	—
+	0	0	$ax^2 = 0$	—	—
+	0	—	$ax^2 = c$	$5x^2 = 80$	metoda 2
+	—	+	$ax^2 + c = bx$	$x^2 + 21 = 10x$	metoda 3
+	—	0	$ax^2 = bx$	$x^2 = 5x$	metoda 4
+	—	—	$ax^2 = bx + c$	$x^2 = 3x + 4$	metoda 5

Metoda 1: Metodę tę ilustruje następujący rysunek:

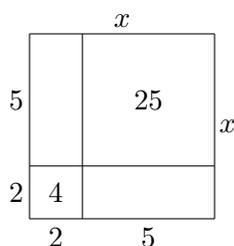


Pole dużego kwadratu wynosi $39 + 25 = 64 = 8 \times 8$. Stąd $x + 5 = 8$, czyli $x = 3$.

Metoda 2: Metoda ta jest bardzo prosta. Skoro $5x^2 = 80$, to wiemy, że $x^2 = \frac{80}{5} = 16$, czyli $x = 4$.



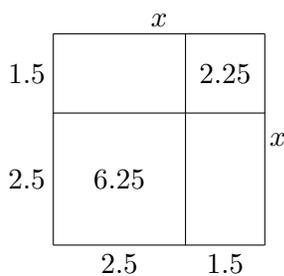
Metoda 3: Metoda ta stanowi, w pewnym sensie, odwrócenie metody 1. Poniższy rysunek ilustruje jej zastosowanie do przykładowego równania:



Pole dużego kwadratu wynosi x^2 . Wycinamy z niego dwa prostokąty o wymiarach $5 \times x$, przy czym na obszarze kwadratu 5×5 (o powierzchni 25) pokrywają się one. Tak więc mały kwadracik ma powierzchnię $x^2 - 10x + 25 = x^2 - 10x + 21 + 4 = 4 = 2^2$. Stąd $x = 5 + 2 = 7$.

Metoda 4: Dla równań postaci $ax^2 = bx$ jasno widać, że jedyne rozwiązanie tego równania (oprócz $x = 0$) to $x = \frac{b}{a}$. Tak więc dla $x^2 = 5x$ otrzymujemy $x = 5$.

Metoda 5: Metoda ta jest analogiczna do metod 1 i 3. Poniższy rysunek ilustruje jej zastosowanie do przykładowego równania:



Pole dużego kwadratu wynosi x^2 . Wycinamy z niego dwa prostokąty o wymiarach $1.5 \times x$, przy czym na obszarze kwadratu 1.5×1.5 (o powierzchni 2.25) pokrywają się one. Tak więc mały kwadracik ma powierzchnię $x^2 - 3x + 2.25 = 4 + 2.25 = 6.25 = 2.5^2$. Stąd $x = 1.5 + 2.5 = 4$.

1.4 Algorytm Euklidesa

Euklides (*Εὐκλείδης*), grecki filozof i matematyk działający na przełomie IV i III w p.n.e. w Aleksandrii. Jego największym dziełem były *Elementy* — kompendium ówczesnej wiedzy matematycznej, znane głównie z opisu geometrii. *Elementy* zawierają również wyniki dotyczące teorii liczb, w tym najstarszy chyba znany algorytm — algorytm Euklidesa na obliczanie największego wspólnego dzielnika dwóch liczb.

Algorytmem tym zajmiemy się bliżej na jednym z późniejszych wykładów. Teraz jedynie przytoczymy prostszą wersję tego algorytmu (opartą na odejmowaniu), jako przykład algorytmu. Algorytm ten opiera się na następującej obserwacji:

$$NWD(x, y) = NWD(x - y, y)$$

Algorytm Euklidesa Dane: dodatnie liczby całkowite x i y . Wynik: $NWD(x, y)$.

1. Dopóki $x \neq y$ powtarzaj następujący krok:
2. Od większej z liczb x i y odejmij mniejszą.
3. Wynik to $x = y = NWD(x, y)$.

1.5 Algorytm Archimedesza przybliżania liczby π

Archimedes ($\acute{\text{A}}\rho\chi\iota\mu\acute{\eta}\delta\eta\sigma$, ok. 287–212 p.n.e.) wszechstronny grecki filozof. Większość życia spędził w Syrakuzach, gdzie się urodził i zginął w trakcie zdobywania miasta przez Rzymian po dwóch latach oblężenia (druga wojna punicka).

Archimedes opracował algorytm przybliżania liczby π . Liczba π jest równa proporcji obwodu okręgu do jego średnicy. Należy więc przybliżyć obwód okręgu o średnicy jednostkowej. Archimedes zauważył, że jeżeli w okrąg wpisujemy wielokąt foremny i równocześnie opiszemy na tym okręgu taki sam wielokąt, to obwód okręgu będzie mniejszy od obwodu wielokąta opisanego, ale większy, od wielokąta wpisanego. Równocześnie, jeżeli będziemy zwiększać liczbę boków wielokątów, to obwód wielokąta opisanego będzie malał, a wpisanego rósł. Dla sześciokąta uzyskujemy proste oszacowanie:

$$3 < \pi < 2\sqrt{3}$$

Powiedzmy, że n -kąt foremny wpisany w okrąg jednostkowy ma boki długości x , a n -kąt foremny opisany na tymże okręgu ma boki długości X . Zastanówmy się, jakiej długości będą boki $2n$ -kątów wpisanego i opisanego na okręgu jednostkowym? Oznaczmy je odpowiednio przez x' i X' . Możemy je wyprowadzić korzystając z twierdzenia Pitagorasa (ok. 570–495 p.n.e). Oznaczmy:

$$y = \sqrt{1 - \frac{x^2}{4}}$$

Z podobieństwa odpowiednich trójkątów otrzymujemy:

$$X = \frac{x}{y} = \frac{2x}{\sqrt{4 - x^2}}$$

Ponadto:

$$\begin{aligned} x' &= \sqrt{\frac{x^2}{4} + (1 - y)^2} = \sqrt{\frac{x^2}{4} + 1 + y^2 - 2y} = \\ &= \sqrt{\frac{x^2}{4} + 1 + 1 - \frac{x^2}{4} - 2\sqrt{1 - \frac{x^2}{4}}} = \sqrt{2 - \sqrt{4 - x^2}} = \\ &= \sqrt{2 - \frac{2x}{X}} \end{aligned}$$


```

\newcounter{i}
\newcounter{a}
\newcounter{b}
\newcounter{c}

\newcommand{\fibtab}[1]{
  \def\heads{1|}
  \def\inds{$i$}
  \def\wyniki{$Fib_i$}
  \setcounter{i}{0}
  \setcounter{a}{0}
  \setcounter{b}{1}
  \@whilenum\value{i}<#1\do {
    \addtocounter{i}{1}
    \edef\heads{\heads 1|}
    \edef\inds{\inds & \thei}
    \edef\wyniki{\wyniki & \theb}
    \setcounter{c}{\value{a}}
    \addtocounter{c}{\value{b}}
    \setcounter{a}{\value{b}}
    \setcounter{b}{\value{c}}
  }
  \begin{tabular}{\heads}
    \hline \inds \\
    \hline \wyniki \\
    \hline
  \end{tabular}
  \fibtab{12}
}

```

i	1	2	3	4	5	6	7	8	9	10	11	12
Fib_i	1	1	2	3	5	8	13	21	34	55	89	144

Przykład ten dowodzi, że dowolny edytor lub system składu tekstów wyposażony w odpowiednio silny mechanizm makrodefinicji staje się językiem programowania.

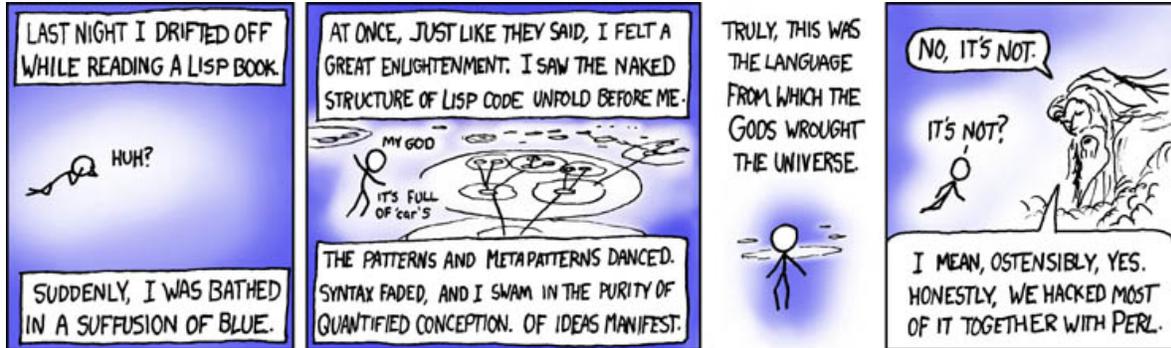
- Opisy skalowalnych czcionek (np. MetaFont):
 - opis czcionek — program,
 - generowanie wyglądu czcionek przy zadanej rozdzielczości i dla danego urządzenia — wykonanie,
 - wygląd czcionek — efekt.

1.7 Dziedzina algorytmiczna

Opisując algorytm, używamy pewnych pojęć elementarnych. W przypadku algorytmu Al-Chwarizmiego wykorzystujemy operacje na liczbach i relacje między nimi. Obecnie użylibyśmy arytmetyki liczb rzeczywistych $\langle R, +, -, *, /, \sqrt{}, <, = \rangle$. W ogólności możemy mieć kilka zbiorów elementarnych wartości, np. liczby całkowite, rzeczywiste, znaki, wartości logiczne.

Dziedzina algorytmiczna to: (zestaw) zbiorów elementarnych wartości, oraz zestaw funkcji (częściowych) i relacji operujących na tych zbiorach. Jeżeli wśród tych zbiorów mamy zbiór wartości logicznych { prawda, fałsz }, to relacje możemy utożsamić z funkcjami w zbiór wartości logicznych.

1.8 Deser



<http://xkcd.com/224/>

Ćwiczenia

W poniższych zadaniach zdefiniuj opisane funkcje matematyczne. Możesz, a nawet powinieneś, zdefiniować przydatne funkcje pomocnicze. Opisz też sposób reprezentacji danych (np. punkt możemy reprezentować jako parę współrzędnych).

1. Wyznacz przecięcie dwóch prostokątów o bokach równoległych do osi współrzędnych i całkowitych współrzędnych wierzchołków.
2. Zdefiniuj funkcję określającą, czy dwa odcinki o końcach o współrzędnych całkowitych przecinają się. (Dostępna jest tylko dziedzina algorytmiczna liczb całkowitych i wartości logicznych.)
3. Napisz funkcję określającą, czy dwa równoległoboki mają niepuste przecięcie.
4. Czy punkt leży wewnątrz wielokąta (niekoniecznie wypukłego, ale bez samoprzecięć). Wielokąt jest dany w postaci ciągu kolejnych wierzchołków na obwodzie (w kolejności przeciwnej do ruchu wskazówek).

Wytyczne dla prowadzących ćwiczenia

Na tych ćwiczeniach nie mówimy nic o języku programowania — ani o Ocamlu, ani o Pascalu. W szczególności, dla studentów nie powinno mieć znaczenia, czy są z potoku funkcyjnego, czy imperatywnego. Rozwiązania zadań powinny mieć postać definicji funkcji matematycznych. Nie należy przesadzać z formalnością. Należy zwrócić uwagę na:

- Reprezentację danych. W treściach zadań nie jest określony sposób reprezentacji punktów, odcinków, prostokątów itp. Wybór reprezentacji jest prosty, ale należy go dokonać. Czasami ma istotny wpływ na długość rozwiązania. Pytanie, co z przypadkami "zdegenerowanymi"?
- Znaczące nazwy funkcji, argumentów i dziedzin.
- Podział problemu na podproblemy, czyli jakie funkcje pomocnicze będą powstawać. Dobrym podziałem odpowiadają prostsze rozwiązania. W trakcie rozwiązywania można dyskretnie nadzorować ten proces, a na koniec ćwiczeń należy na to zwrócić uwagę.

Uwagi dotyczące zadań:

Ad. 1 Punkt to para współrzędnych. Prostokąt to para jego wierzchołków (dolny lewy i górny prawy) lub para zakresów – taki sposób reprezentacji jest najbardziej intuicyjny, choć wcale nie daje najprostszego rozwiązania. Alternatywnie, prostokąt, to para przedziałów – jego rzutów na osie. Należy zwrócić uwagę na to, że przecięcie dwóch prostokątów może być puste. Najlepiej przyjąć, że prostokąty zdegenerowane są dopuszczalne i reprezentują pusty zbiór.

Podział na podproblemy:

- przecięcie dwóch prostokątów,
- rzutowanie,
- przecięcie odcinków na osi,
- budowa prostokąta na podstawie jego rzutów.

Ad. 2 Odcinek to para punktów, wektor swobodny to punkt. Dwa odcinki się przecinają, jeżeli końce każdego z nich leżą po przeciwnych stronach prostej wyznaczonej przez drugi z nich. Po której stronie prostej leży punkt – iloczyn wektorowy. Uwaga na odcinki zdegenerowane i przypadki brzegowe. Jako dodatkowe kryterium pomoże przecinanie się prostokątów, których dane odcinki są przekątnymi.

Możliwy podział na podproblemy:

- czy dwa odcinki się przecinają?
- czy prosta (wyznaczona przez dwa punkty) przecina odcinek?
- iloczyn wektorowy,
- konwersja pary punktów na wektor swobodny,
- konwersja przekątnej (odcinka) na prostokąt,
- przecięcie dwóch prostokątów,
- czy prostokąt jest niepusty?

Ad. 3 To zadanie jest opcjonalne. Równoległobok to punkt i dwa wektory swobodne. Równoległoboki się przecinają, jeżeli:

- jeden z wierzchołków jednego z równoległoboków leży wewnątrz drugiego równoległoboku, lub
- ich obwody się przecinają.

Możliwy podział na podproblemy:

- czy równoległoboki się przecinają?
- suma wektorów i przesuwanie punktów,
- czy punkt jest wewnątrz równoległoboku?
- iloczyn wektorowy,
- czy obwody dwóch równoległoboków się przecinają?
- czy odcinek przecina obwód równoległoboku?
- czy dwa odcinki się przecinają?

Uwaga: To zadanie jest technicznie upierdliwe, a przez to dosyć pracochłonne. Z jednej strony, jego rozwiązanie potrafi zająć sporo czasu. Z drugiej, nie należy przesadzać z formalizmem.

Ad. 4 Należy rozważyć prostą poziomą przechodzącą przez dany punkt oraz boki wielokąta, które przecina. Jeżeli po jednej (lewej lub prawej) stronie punktu jest nieparzysta ich liczba, to punkt leży wewnątrz wielokąta. Uwaga na przypadki graniczne: gdy dany punkt leży na obwodzie wielokąta lub opisana prosta przecina wierzchołek wielokąta.

Wykład 2. Podstawy języka programowania

W **każdym** języku programowania mamy trzy rodzaje konstrukcji językowych:

- podstawowe symbole (typy, wartości, operacje, relacje itp.) — pochodzące z dziedziny algorytmicznej,
- sposoby konstrukcji — czyli jak z prostszych całości budować bardziej skomplikowane,
- sposoby abstrakcji — czyli jak złożone konstrukcje mogą być nazwane i dalej wykorzystywane tak, jak podstawowe elementy.

Nasza dziedzina algorytmiczna zawiera m.in.:

- typy: `bool`, `int`, `float`, `char`, `string`,
- stałe: logiczne (`true` i `false`), całkowite (np.: 0, 1, -2), rzeczywiste (np.: 2.3, -3.4, $4.5E-7$), znakowe (np.: 'a', napisy (np. "ala ma kota").
- procedury¹: `+`, `-`, `*`, `/`, `mod`, `+`, `-`, `*`, `/`, `||`, `&&`, `not`, `=`, `≤`, `≥`, `<`, `>`, `<>`, `^`.

Powtórka: rozróżnienie symboli od ich interpretacji.

2.1 BNF

W kolejnych wykładach będziemy przedstawiać rozmaite konstrukcje językowe Ocaml'a. Do opisu składni tych konstrukcji, oprócz języka nieformalnego i przykładów, będziemy używać notacji BNF (ang. Backus-Naur form). Jest to rozszerzenie gramatyk bezkontekstowych, poręczne przy opisywaniu składni języków programowania. Notacja ta ma następującą postać:

- W BNF-ie definiujemy możliwe postaci rozmaitych *konstrukcji składniowych*. Konstrukcje te mają nazwy postaci: $\langle \text{konstrukcja} \rangle$. Odpowiadają one nieterminalom w gramatykach bezkontekstowych.
- Opis składni składa się z szeregu reguł postaci:

$$\langle \text{konstrukcja} \rangle ::= \text{możliwa\ postać}$$

Reguły te opisują możliwe postaci poszczególnych konstrukcji składniowych. Jeżeli dla danej konstrukcji mamy wiele reguł, to traktujemy je jak alternatywy. Reguły te odpowiadają produkcjom w gramatykach bezkontekstowych.

- Po prawej stronie reguł możemy używać następujących konstrukcji:
 - podając tekst, który ma się pojawić dosłownie, będziemy go podkreślać, na przykład: słowo kluczowe,
 - nazw konstrukcji składniowych postaci $\langle \text{konstrukcja} \rangle$ możemy używać również po prawej stronie reguł, na oznaczenie miejsc, w których się one pojawiają,
 - chcąc opisać alternatywne postaci danej konstrukcji, używamy pionowej kreski, $\dots | \dots$,

¹ Od tej pory będziemy używać słowa *funkcja* na określenie pojęcia matematycznego, a słowa *procedura* na określenie pojęcia programistycznego.

- tekst umieszczony w kwadratowych nawiasach jest opcjonalny, [...],
- dodatkowo, możemy używać nawiasów (wąsatych), {...},
- tekst umieszczony w nawiasach postaci {...}* może się powtarzać zero, jeden lub więcej razy,
- tekst umieszczony w nawiasach postaci {...}⁺ może się powtarzać jeden lub więcej razy.

Tego formalizmu będziemy używać do opisu składni.

Przykład: BNF może być użyty do opisywania składni najrozmaitszych rzeczy. Oto opis składni adresów pocztowych:

```

<adres>          ::= <adresat> _ <adres lokalny> _ <adres miasta> [ _ <adres kraju> ]
<adresat>        ::= [ W.P. | Sz.Pan. ] <napis>
<adres lokalny> ::= <ulica> { <numer> [ <litera> ] | <numer> / <numer> } [ m <numer> | / <numer> ]
<adres miasta>  ::= [ <kod> ] <napis>
<adres kraju>   ::= <napis>
<kod>           ::= <cyfra> <cyfra> _ <cyfra> <cyfra> <cyfra>
<cyfra>         ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<numer>         ::= <cyfra> [ <numer> ]

```

Specyfikacja ta oczywiście nie jest kompletna, gdyż nie definiuje czym jest napis.

2.2 Przyrostowy tryb pracy

Kompilator Ocamlu działa w sposób inkrementacyjny, tzn. działa w cyklu powtarzając następujące czynności:

- wczytuje fragment programu,
- kompiluje go, dołączając do już skompilowanych fragmentów,
- wykonuje wprowadzony fragment.

Nazwa “tryb przyrostowy” bierze się stąd, że kompilator nie musi generować całego kodu od razu, lecz kod ten przyrasta z każdym cyklem.

Taki fragment programu, wprowadzany i kompilowany w jednym cyklu, będziemy nazywać *jednostką kompilacji*. Wykonanie jednostki kompilacji może zarówno powodować obliczenia wartości, jak i definiowanie nowych pojęć. Każda jednostka kompilacji musi być w Ocamlu zakończona przez ; ;.

$$\langle \text{program} \rangle ::= \{ \langle \text{jednostka kompilacji} \rangle \underline{; ;} \}^*$$

2.3 Wyrażenia

Najprostsza jednostka kompilacji i podstawowa konstrukcja programistyczna to wyrażenia. Wyrażenia budujemy w standardowy sposób za pomocą stałych, procedur i nawiasów. Jedynym odstępstwem jest to, że argumenty procedur nie muszą być objęte nawiasami i podzielane przecinkami. Operacje, które standardowo zapisujemy infiksowo (np. operacje arytmetyczne) mają również postać infiksową. Rozróżniamy operacje arytmetyczne na liczbach całkowitych i rzeczywistych — te ostatnie mają dodaną kropkę.

Przykład:

```
42;;
- : int = 42

36 + 6;;
- : int = 42

3 * 14;;
- : int = 42

100 - 58;;
- : int = 42

1 * 2 * 3 * 4 * 5 - ((6 + 7) * 8 * 9 / 12);;
- : int = 42

silnia 7 / silnia 5;;
- : int = 42

596.4 /. 14.2;;
- : float = 42.

"ala" ^ " ma " ^ "kota";;
- : string = "ala ma kota"
```

Wszystko, czego potrzeba do budowy wyrażeń, to:

- stałe (liczbowe, napisy lub nazwy stałych),
- zastosowanie procedury do argumentów,
- nawiasów.

Zauważmy, że procedury i ich argumenty są tu traktowane na równi — takie symbole jak `+` czy `*` to po prostu nazwy stałych, których wartościami są procedury wbudowane w język programowania, a `123`, `486.5`, czy `"ala"` to stałe.

2.3.1 Składnia wyrażeń

```

<jednostka kompilacji> ::= <wyrażenie> | ...
<wyrażenie>           ::= ( <wyrażenie> ) | { <wyrażenie> }+ |
                           <operator unarny> <wyrażenie> |
                           <wyrażenie> <operator binarny> <wyrażenie> |
                           <identyfikator> | <stała całkowita> |
                           <stała zmiennopozycyjna> | <stała napisowa> | ...

<operator unarny>      ::= - | not | ...
<operator binarny>    ::= + | +. | - | -. | * | *. | / | /. | mod | || | && |
                           = | ≤ | ≤= | ≥ | ≥= | ^ | ...

```

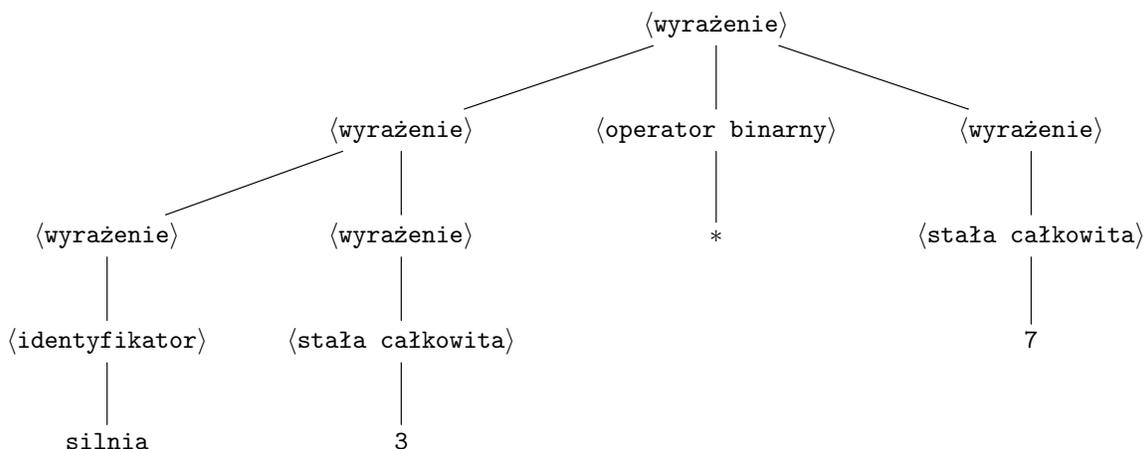
Zastosowania procedur wiążą najsilniej, operatory unarne słabiej, a operatory binarne najsłabiej, przy czym zachowana jest tradycyjna kolejność wiązania operacji arytmetycznych.

Uwaga: Powyższy opis składni nie jest jednoznaczny, tzn. ten sam program może zostać rozłożony składniowo na więcej niż jeden sposób (ale dzięki temu jest bardziej czytelny). Nie będziemy aż tak formalni. Zamiast tego ewentualne niejednoznaczności będziemy wyjaśniać w sposób opisowy.

2.3.2 Obliczanie wartości wyrażeń

Efekt „działania” wyrażenia jest jego wartość. Wyrażenie możemy sobie przedstawić w postaci drzewa (tzw. drzewo wyprowadzenia). W liściach drzewa mamy stałe, a węzły wewnętrzne to procedury. Nawiasy wraz z priorytetami operatorów wyznaczają kształt takiego drzewa.

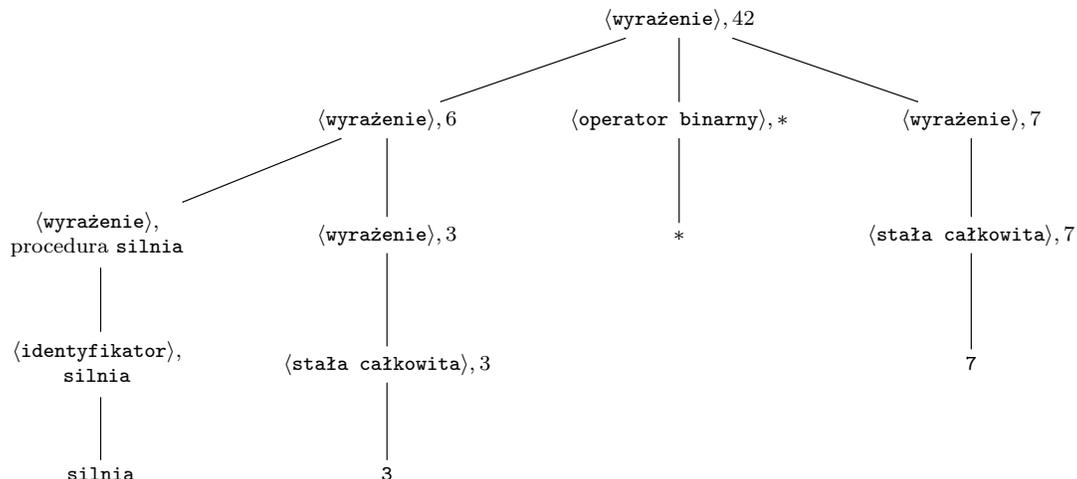
Przykład: Oto drzewo wyprowadzenia dla wyrażenia silnia 3 * 7:



Wszystkie nazwane stałe są przechowywane w tzw. *środowisku*. Przyporządkowuje ono nazwom ich wartości. Jeśli w wyrażeniu występują nazwy stałych, to ich wartości są pobierane ze środowiska.

Wyliczenie wartości wyrażenia możemy sobie wyobrazić jako udekorowanie drzewa wyprowadzenia wartościami, od liści do korzenia. Wartość w korzeniu to wynik.

Przykład: Oto powyższe drzewo wyprowadzenia udekorowane wartościami:



2.4 Definicje stałych

Kolejną podstawową jednostką kompilacji jest definicja stałej.

$$\begin{aligned} \langle \text{jednostka kompilacji} \rangle &::= \langle \text{definicja} \rangle | \dots \\ \langle \text{definicja} \rangle &::= \underline{\text{let}} \langle \text{identyfikator} \rangle \equiv \langle \text{wyrażenie} \rangle | \dots \end{aligned}$$

Definicji stałej nie da się wyrazić jako procedury ani wyrażenia, ponieważ zmienia ona środowisko. Dlatego też `let` jest nazywane *formą specjalną*. „Obliczenie” definicji wykonywane jest następująco:

- oblicz wartość wyrażenia tworzącego definicję,
- do środowiska wstawiany jest identyfikator, któremu przyporządkowywana jest wartość obliczonego wyrażenia.

Jeżeli symbol jest już w środowisku, to można *przysłonić* go i nadać symbolowi nowe znaczenie.

Uwaga: To jest uproszczony model. W miarę jak będziemy poznawać kolejne konstrukcje językowe, będziemy go w miarę potrzeb rozszerzać.

Przykład:

```
let a = 4;;
val a : int = 4
```

```
let b = 5 * a;;
val b : int = 20
```

```
let pi = 3.14;;
val pi : float = 3.14
```

```
pi *. 4.0 *. 4.0;;  
val - : float = 50.24
```

```
let a = "ala";;  
val a : string = "ala"
```

Możliwe jest też równoczesne definiowanie wielu stałych. W takim przypadku najpierw obliczane są wszystkie wyrażenia, a następnie ich wartości są przypisywane w środowisku odpowiednim identyfikatorom.

$$\langle \text{definicja} \rangle ::= \underline{\text{let}} \langle \text{identyfikator} \rangle = \langle \text{wyrażenie} \rangle \\ \{ \underline{\text{and}} \langle \text{identyfikator} \rangle = \langle \text{wyrażenie} \rangle \}^*$$

Przykład:

```
let a = 4 and b = 5;;  
val a : int = 4  
val b : int = 5
```

```
let a = a + b and b = a * b;;  
val a : int = 9  
val b : int = 20
```

2.5 Wartości logiczne i wyrażenia warunkowe.

Wartości typu `bool` to wartości logiczne. Składają się na nie dwie stałe: `true` i `false`. Z wartościami logicznymi związane są wyrażenia warunkowe postaci:

$$\langle \text{wyrażenie} \rangle ::= \underline{\text{if}} \langle \text{wyrażenie} \rangle \underline{\text{then}} \langle \text{wyrażenie} \rangle \underline{\text{else}} \langle \text{wyrażenie} \rangle$$

Wartością pierwszego wyrażenia powinna być wartość logiczna, a pozostałe dwa wyrażenia muszą być tego samego (lub uzgadnialnego) typu. Najpierw obliczane jest pierwsze wyrażenie. Jeżeli jest ono równe `true`, to obliczane jest drugie wyrażenie i jego wartość jest wartością całego wyrażenia. Jeżeli jest ono równe `false`, to obliczane jest trzecie wyrażenie i jego wartość jest wartością całego wyrażenia.

Obliczanie wyrażenia `if-then-else` jest w pewnym sensie leniwe — obliczane jest tylko to, co niezbędne. Jeżeli warunek jest prawdziwy, to nie jest obliczany ostatni człon, a jeśli jest fałszywy, to nie jest obliczany drugi człon. Wyrażenie `if-then-else` jest formą specjalną — nie można go zdefiniować jako procedury, gdyż zawsze jeden z argumentów w ogóle nie jest obliczany.

Przykład:

```
let x = 0.0238095238095238082;;  
val x : float = 0.0238095238095238082  
  
if x = 0.0 then 0.0 else 1.0 /. x;;  
- : float = 42.
```

Operatory logiczne `&&` i `||`, to odpowiednio koniunkcja i alternatywa. Są to również formy specjalne, a nie procedury. Są one równoważne odpowiednim wyrażeniom warunkowym:

```
x && y ≡ if x then y else false  
x || y ≡ if x then true else y
```

Negacja, oznaczana przez `not` jest już zwykłą procedurą.

2.6 Definicje procedur

Mamy tylko trzy podstawowe konstrukcje językowe:

- użycie wartości zdefiniowanej w środowisku (nieważne liczby, czy procedury),
- zastosowanie procedur do argumentów (np. obliczenie mnożenia),
- definiowanie nowych wartości.

Jak wcześniej wspomnieliśmy, procedury są obywatelami pierwszej kategorii, czyli równie dobrymi wartościami, jak wszystkie inne. Wynika stąd, że:

- można definiować nazwane procedury (stałe proceduralne),
- procedura może być wartością wyrażenia,
- procedury mogą być argumentami i wynikami procedur (sic!).

Definicje nazwanych procedur mają następującą postać:

$$\langle \text{definicja} \rangle ::= \underline{\text{let}} \langle \text{identyfikator} \rangle \{ \langle \text{identyfikator} \rangle \}^+ \equiv \langle \text{wyrażenie} \rangle$$

Pierwszy z identyfikatorów to nazwa definiowanej procedury, a reszta, to jej *parametry formalne*. Wyrażenie zaś to *treść* procedury. Określa ono wartość procedury, w zależności od wartości argumentów. Zwróćmy uwagę, że argumenty nie są otoczone nawiasami ani oddzielone przecinkami.

Przykład:

```
let abs x =
  if x < 0 then -x else x;;
val abs : int ->int = <fun>

let square x = x *. x;;
val square : float ->float = <fun>

let pow r = pi *. (square r);;
val pow : float ->float = <fun>

pow 4.0;;
- : float = 50.24

let twice x = 2 * x;;
val twice : int ->int = <fun>

twice (twice 3);;
- : int = 12

let mocium s = "mocium panie," ^ s;;
val mocium : string ->string = <fun>

mocium (mocium "me wezwanie");;
- : string = "mocium panie,mocium panie,me wezwanie"
```

Tak jak w przypadku innych wartości, gdy definiujemy procedurę, kompilator odpowiada podając nam „typ” tej procedury. Typ ten zawiera typ argumentu i wyniku połączone strzałką. Na przykład, `int -> int` oznacza, że argument i wynik procedury są liczbami całkowitymi. W przypadku większej liczby argumentów zobaczymy większą liczbę strzałek. Typami procedur zajmiemy się dokładniej, gdy będziemy mówić o procedurach wyższych rzędów.

```
let plus x y = x + y;;
val plus : int ->int ->int = <fun>
```

W momencie, gdy zastosujemy procedurę do jej argumentów, jej wartość jest obliczana w następujący sposób:

- wyznaczana jest procedura, którą należy zastosować i jej argumenty,
- na podstawie środowiska (w którym była definiowana procedura) tworzone jest tymczasowe środowisko, w którym dodatkowo parametrom formalnym są przyporządkowane wartości argumentów,
- w tym tymczasowym środowisku obliczane jest wyrażenie stanowiące treść procedury,
- wartość tak obliczonego wyrażenia jest wartością zastosowania procedury do argumentów.

Uwaga: Procedury nie są funkcjami matematycznymi — mogą być nieokreślone, np. dzielenie przez 0, lub ich obliczanie może się zapętlać.

Przykład: Przykład ten ilustruje, dlaczego treść procedury jest obliczana w środowisku powstałym ze środowiska, w którym dana procedura została zdefiniowana, a nie z aktualnego środowiska.

```
let a = 24;;
let f x = 2 * x + a;;
let a = 15;;
f 9;;
- : int = 42
```

2.6.1 λ -abstrakcja

Możemy też tworzyć procedury bez nazwy, tzn. takie procedury, które są wartością pewnego rodzaju wyrażenia, natomiast nie muszą pojawiać się w środowisku. Procedury takie możemy tworzyć za pomocą tzw. λ -abstrakcji.

Zauważmy, że funkcja, jako pojęcie matematyczne, nie musi mieć nazwy. Zwykle nadajemy funkcjom nazwy, gdyż tak jest poręczniej. Na przykład $x \mapsto x + 1$ jest również dobrym opisem funkcji (której wartość jest o 1 większa od argumentu). λ -abstrakcja ma podobną postać. Składa się ona ze słowa kluczowego `function`, parametru formalnego funkcji oraz wyrażenia określającego jej wartość.

$$\langle \text{wyrażenie} \rangle ::= \text{function } \langle \text{identyfikator} \rangle \text{ -> } \langle \text{wyrażenie} \rangle$$

Wartością takiego wyrażenia jest jednoargumentowa procedura bez nazwy. Na razie możemy przyjąć, że forma specjalna `let` definiująca procedurę z parametrami jest lukrem syntaktycznym pokrywającym wyrażenie `function` i definicję stałej.

Procedury wieloargumentowe zapisujemy w postaci:

```
function x ->function y ->function z ->...
```

Istnieje również skrócona forma zapisu wieloargumentowych λ -abstrakcji. Zamiast słowa kluczowego `function` możemy użyć jednego słowa kluczowego `fun` i podać od razu kilka argumentów. `Fun` jest tylko lukrem syntaktycznym, ale bardzo poręcznym.

$$\langle \text{wyrażenie} \rangle ::= \text{fun } \{ \langle \text{identyfikator} \rangle \}^+ \text{ -> } \langle \text{wyrażenie} \rangle$$

Przykład: Oto alternatywna definicja procedur z poprzedniego przykładu:

```
(function x ->x * (x + 1)) (2 * 3);;
- : int = 42

let abs = function x ->if x <0 then -x else x;;
val abs : int ->int = <fun>
```

```

let square = function x ->x *. x;;
val square : float ->float = <fun>

let pow = function r ->pi *. ((function x ->x *. x) r);;
val pow : float ->float = <fun>

let twice = function x ->2 * x;;
val twice : int ->int = <fun>

let foo x y = x * (y +2);;
val foo : int ->int ->int = <fun>

let foo = function x ->function y ->x * (y +2);;
val foo : int ->int ->int = <fun>

let foo = fun x y ->x * (y +2);;
val foo : int ->int ->int = <fun>

```

2.6.2 Procedury rekurencyjne

Procedury mogą być rekurencyjne. Musimy to jednak jawnie określić dodając słówko `rec`:

$$\langle \text{definicja} \rangle ::= \underline{\text{let}} \underline{\text{rec}} \{ \langle \text{identyfikator} \rangle \}^+ \underline{=} \langle \text{wyrażenie} \rangle$$

Możemy definiować procedury rekurencyjne podając parametry formalne lub używając λ -abstrakcji.

Przykład: Przykłady definicji rekurencyjnych:

```

let rec silnia num =
  if num <2 then 1 else num * silnia (num - 1);;
val silnia : int ->int = <fun>

silnia 7 / silnia 5;;
- : int = 42

let rec fib num =
  if num <2 then num else fib (num - 1) + fib (num - 2);;
val fib : int ->int = <fun>

fib 10 - fib 7;;
- : int = 42

let rec petla x = x + petla x;;
val petla : int ->int = <fun>

let rec p = (function x ->p (x + 1));;
val p : int ->'a = <fun>

```

```
let rec x = x + 2;;
```

This kind of expression is not allowed as right-hand side of 'let rec'

Istnieją inne obiekty rekurencyjne, niż procedury.

Dodanie słówka `rec` powoduje, że definiowana procedura jest obecna w środowisku, w którym później będzie obliczana jej treść. Podobnie jak w przypadku definicji innych wartości, możemy równocześnie definiować kilka procedur, używając `and`. Jest to przydatne, gdy piszemy procedury wzajemnie rekurencyjne.

2.6.3 Rekurencja ogonowa

Jeśli wartości zwracane przez wszystkie wywołania rekurencyjne, bez żadnego przetwarzania są przekazywane jako wynik danej procedury, to mówimy o **rekurencji ogonowej**. Konstrukcje takie jak `if-then-else`, czy inne, które poznamy, a które dotyczą tylko przepływu danych, ale nie ich przetwarzania, nie wpływają na ogonowość rekursji.

Rekurencja ogonowa jest istotna z tego powodu, że potrzebuje mniej pamięci. Każde wywołanie procedury zajmuje pewną ilość pamięci. Jednak ogonowe wywołanie rekurencyjne nie wymaga dodatkowej pamięci, gdyż używa pamięci używanej przez obecne wywołanie.

Analizę rekurencji ogonowej na złożoność pamięciową zajmiemy się w przyszłości, gdy poznamy semantykę operacyjną programów funkcyjnych oraz nauczymy się analizować koszty programów. Na razie skupimy się tylko na tym, jak tworzyć ogonowe procedury rekurencyjne.

Często, gdy chcemy użyć rekurencji ogonowej, musimy zdefiniować dodatkową rekurencyjną procedurę pomocniczą, posiadającą dodatkowy(-e) argument(-y). Jeden z takich dodatkowych argumentów to stopniowo konstruowany wynik. Argument ten jest zwyczajowo nazywany *akumulatorem*.

Przykład: Silnia z akumulatorem:

```
let rec silnia_pom a n =
  if n <= 1 then a else silnia_pom (a * n) (n - 1);;
val silnia_pom : int ->int ->int = <fun>
```

```
let silnia num = silnia_pom 1 num;;
val silnia : int ->int = <fun>
```

```
silnia 3;;
- : int = 6
```

Przykład: Liczby Fibonacciego z akumulatorem:

```
let rec fib_pom a b num =
  if num = 0 then a else fib_pom b (a + b) (num - 1);;
val fib_pom : int ->int ->int ->int = <fun>
```

```
let fib num = fib_pom 0 1 num;;
```

```
val fib : int ->int = <fun>
```

```
fib 5;;  
- : int = 5
```

2.7 Definicje lokalne

Jeśli chcemy zdefiniować jakąś wartość lokalnie, używamy następującej postaci wyrażenia:

$$\langle \text{wyrażenie} \rangle ::= \langle \text{definicja} \rangle \text{ in } \langle \text{wyrażenie} \rangle$$

Zakres definicji jest ograniczony do wyrażenia po in.

Przykład:

```
let pow x =  
  let pi = 3.14  
  and s = x *. x  
  in pi *. s;;  
val pow : float ->float = <fun>
```

```
pow 4.0;;  
- : float = 50.24
```

```
let pitagoras a b =  
  let square x = x * x  
  in  
  square a + square b;;  
val pitagoras : int ->int ->int = <fun>
```

```
pitagoras 3 4;;  
- : int = 25
```

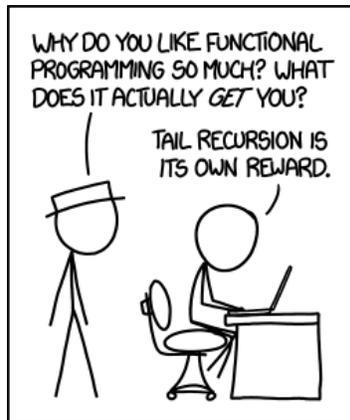
Zagadka: Formę specjalną let-in można, do pewnego stopnia, zastępować λ -abstrakcją. W jaki sposób?

2.8 Komentarze

Komentarze w Ocamlu umieszczamy w nawiasach postaci (* ...*). Komentarze mogą pojawiać się wszędzie tam, gdzie mogą występować białe znaki. Komentarze można zagnieżdżać.

```
(* To jest komentarz. *)  
(* To też jest komentarz (* ale zagnieżdżony *)). *)
```

2.9 Deser



<http://xkcd.com/1270/>

Ćwiczenia

Rozwiązania poniższych zadań to proste procedury operujące na liczbach całkowitych. Dla każdej procedury rekurencyjnej podaj jej specyfikację, tj. warunek początkowy i końcowy, oraz uzasadnij jej poprawność. Poprawność procedur rekurencyjnych można pokazać przez indukcję.

1. Stopień parzystości liczby całkowitej x to największa taka liczba naturalna i , że x dzieli się przez 2^i . Liczby nieparzyste mają stopień parzystości 0, liczby 2 i -6 mają stopień parzystości 1, a liczby 4 i 12 mają stopień parzystości 2. Przyjmujemy, że 0 ma stopień parzystości -1 .
Napisz procedurę **parzystość** wyznaczającą stopień parzystości danej liczby całkowitej.
2. Napisz procedurę, która przekształca daną liczbę naturalną w taką, w której cyfry występują w odwrotnej kolejności, np. 1234 jest przekształcane na 4321.
3. Sumy kolejnych liczb nieparzystych dają kwadraty kolejnych liczb naturalnych, zgodnie ze wzorem: $\sum_{i=1}^k (2i - 1) = k^2$. Wykorzystaj ten fakt do napisania procedury **sqrt** obliczającej **sqrt** $x = \lfloor \sqrt{x} \rfloor$ i nie korzystającej z mnożenia ani dzielenia.
4. Napisz procedurę, która sprawdza, czy dana liczba jest pierwsza.
5. Liczbę naturalną nazwiemy *rzadką*, jeżeli w jej zapisie binarnym żadne dwie jedynki nie stoją obok siebie. Napisz procedurę **rzadkie** : **int** \rightarrow **int**, która dla danej liczby naturalnej n zwróci najmniejszą rzadką liczbę naturalną większą od n .
Na przykład, dla $n = 42 = 101010_2$ mamy **rzadkie** $42 = 1000000_2 = 64$.
6. Liczbę naturalną nazwiemy *rzadką*, jeżeli w jej zapisie binarnym żadne dwie jedynki nie stoją obok siebie. Napisz procedurę **rzadkie** : **int** \rightarrow **int**, która dla danej liczby naturalnej n wyznaczy liczbę dodatnich liczb rzadkich, które nie przekraczają n .
Na przykład, dla $n = 42 = 101010_2$ mamy **rzadkie** $42 = 20$.
7. Napisz procedurę, która sprawdza, czy dana liczba jest podzielna przez 9 w następujący sposób: jedyne liczby jednocyfrowe podzielne przez 9 to 9 i 0; reszta z dzielenia liczby wielocyfrowej przez 9 jest taka sama, jak reszta z dzielenia sumy jej cyfr przez 9; jeśli suma cyfr jest wielocyfrowa, to całość powtarzamy, aż do uzyskania liczby jednocyfrowej.
8. Napisz procedurę, która sprawdza, czy dana liczba jest podzielna przez 11 w następujący sposób: sumujemy cyfry liczby znajdujące się na parzystych pozycjach oraz te na nieparzystych pozycjach, różnica tych dwóch liczb przystaje modulo 11 do wyjściowej liczby; krok ten należy powtarzać aż do uzyskania liczby jednocyfrowej.
9. Zaimplementuj kodowanie par liczb naturalnych jako liczby naturalne. To znaczy, napisz procedurę dwuargumentową, która koduje dwie liczby dane jako argumenty w jednej liczbie naturalnej. Dodatkowo napisz dwie procedury, które wydobywają z zakodowanej pary odpowiednio pierwszą i drugą liczbę.
10. Napisz procedurę, która dla danej liczby n sprawdzi czy pierścień reszt modulo n zawiera nietrywialne pierwiastki z 1 (tj. takie liczby k , $k \neq 1$, $k \neq n-1$, że $k^2 \equiv 1 \pmod{n}$). Nota bene, jeśli takie pierwiastki istnieją, to liczba n nie jest pierwsza. Odwrotna implikacja jednak nie zachodzi — np. dla $n = 4$ nie ma nietrywialnych pierwiastków z 1.

Wytyczne dla prowadzących ćwiczenia

Przy rozwiązywaniu zadań kładziemy nacisk na poprawność, a nie kładziemy na ramię nacisku na efektywność. W szczególności, nie kładziemy nacisku na stosowanie rekurencji ogonowej, choć jej nie zabraniamy. Każda procedura powinna być wyspecyfikowana. Jeżeli weryfikacja (następny wykład) nie została jeszcze omówiona, to uzasadnienie poprawności może być mniej formalne. Jeśli zaś była omówiona, to stosujemy przedstawioną na wykładzie metodologię. (W przypadku procedur rekurencyjnych dowodzimy ich poprawności przez indukcję i podajemy funkcję miary dowodzącą poprawności definicji rekurencyjnych. Należy zwrócić uwagę na pojawiające się niezmienniki.)

Wykład 3. Dekompozycja problemu, weryfikacja rozwiązania

3.1 Specyfikacja problemu

- Specyfikacja = opis celu = wartość/funkcja częściowa opisana matematycznie.
- Implementacja = realizacja w języku programowania.
- Specyfikacja może dopuszczać wiele implementacji, np.: $\forall x : (fx) * (fx) = x * x$ czyli $|fx| = |x|$.
- Jeśli specyfikacja jest funkcją częściową, to implementacja może mieć większą dziedzinę.

$$\forall x : x > 0 \Rightarrow \dots$$

```
let f x = if x > 0 then ... else ...
```

- Często specyfikację dzielimy na dwie części:
 - **warunek początkowy** opisuje wszystkie dozwolone wartości argumentów; interesują nas wartości wyników wyłącznie dla argumentów spełniających warunek początkowy,
 - **warunek końcowy** opisuje w jaki sposób wyniki zależą od argumentów; dla argumentów spełniających warunek początkowy muszą istnieć wyniki spełniające warunek końcowy.
- To, że program spełnia specyfikację można wyrazić formułą:

$$\text{warunek początkowy(dane)} \implies \text{warunek końcowy(dane, wynik)}$$

- Pełna i częściowa poprawność.
- Tworząc procedury pomocnicze powinniśmy określić również ich specyfikacje. W przypadku iteracyjnych procedur rekurencyjnych są to niezmienniki iteracji.
- Weryfikacja procedur pomocniczych przypomina dowody lematów.
- Schemat dowodzenia własności stopu:
 - w przypadku procedur rekurencyjnych należy udowodnić, że ich wywołania (spełniające warunki początkowe) nie zapętłają się,
 - określamy funkcję miary, zależną od argumentów, która przyjmuje wartości naturalne; jest to górne ograniczenie na głębokość rekurencji, dowód przebiega indukcyjnie (po wartościach funkcji miary),
 - zakładamy, że argumenty wywołania spełniają warunek początkowy, dla danych spełniających warunek początkowy wartość funkcji miary jest nieujemna,
 - pokazujemy, że wywołania procedur spełniają odpowiednie warunki początkowe,
 - dla wszystkich wywołań rekurencyjnych pokazujemy, że funkcja miary maleje.
- Schemat dowodzenia częściowej poprawności:
 - zakładamy, że argumenty spełniają warunek początkowy, a wszystkie wywołania procedur (łącznie z rekurencyjnymi) spełniają specyfikację,
 - pokazujemy, że wynik jest zgodny z warunkiem końcowym.

3.2 Algorytm Euklidesa (wersja z odejmowaniem)

Specyfikacja: dla $x, y > 0$ mamy $(\text{nwd } x \ y) = \text{NWD}(x, y)$. implementacja:

```
let rec nwd x y =
  if x = y then x else
  if x > y then
    nwd (x - y) y
  else
    nwd x (y - x);;
```

A co się dzieje jeżeli jeden z argumentów jest równy 0? Nowy rodzaj nieokreśloności — zapętlenie. Pojęcia częściowej i całkowitej poprawności — w przypadku częściowej poprawności implementacja może być mniej określona niż specyfikacja.

Dowód: Funkcja miary $f = \max(x, y)$. Jeśli $x = y$, to oczywiste. Załóżmy, że $x > y$. Dla każdego d takiego, że $d|x$ i $d|y$ mamy $d|(x - y)$, a więc $\text{nwd}(x, y) = \text{nwd}(x - y, y)$. Dla $x < y$ analogicznie. \square

Weryfikacja `nwd` — łatwo wykazać własność stopu, a trudniej poprawność.

3.3 Silnia i przykład niezmiennika

Oto jedna z możliwych definicji procedury obliczającej silnię, z rekurencją ogonową.

```
let silnia n =
  let rec s a k =
    if k = n then a
    else s (a * (k+1)) (k+1)
  in s 1 1;;
```

Specyfikacja procedury `silnia` jest dosyć naturalna. Dla $x > 0$ mamy $\text{silnia } x = x!$. Jaka jest jednak specyfikacja procedury pomocniczej `s`? W warunku początkowym należy ująć niezmiennik iteracji:

$$1 \leq k \leq n \wedge a = k!$$

Niezmiennik ten jest spełniony na początku, dla $a = 1$ i $k = 1$. Z każdym krokiem iteracji, zwiększając k zwiększamy odpowiednio a . Na koniec, dla $k = n$ mamy $a = n!$, co daje warunek końcowy `s a k = n!`.

3.4 Szukanie pierwiastka całkowitego przez bisekcję

Mając daną liczbę naturalną num chcemy obliczyć $\lfloor \sqrt{\text{num}} \rfloor$. Warunek początkowy to $\text{num} \geq 0$, a końcowy to `sqrt num = $\lfloor \sqrt{\text{num}} \rfloor$` .

Jedną z metod, jakie możemy zastosować to bisekcja. Jest to proces iteracyjny, w którym zawężamy przedział, do którego należy wynik. Zaczynamy od przedziału $[0; \text{num}]$ korzystając tu z faktu, że $0 \leq \lfloor \sqrt{\text{num}} \rfloor \leq \text{num}$. Przedział zawężamy w każdym kroku o połowę, podnosząc jego środek do kwadratu i porównując z num . Po zawężeniu do pojedynczej liczby całkowitej otrzymujemy wynik.

W warunku początkowym procedury pomocniczej `bisekcja` musimy zawrzeć niezmiennik tej iteracji: $\text{lewy} \leq \sqrt{\text{num}} < \text{prawy} + 1$. Na początku jest on trywialnie spełniony dla $\text{lewy} = 0$

i $prawy = num$. W momencie, gdy $lewy = prawy$ daje nam to natychmiast warunek końcowy: $bisekcja\ lewy\ prawy = \sqrt{num}$.

```
let sqrt num =
  let rec bisekcja lewy prawy =
    if lewy = prawy then lewy
    else
      let srodek = (lewy+prawy) / 2
      in
        if square (srodek + 1) > num then
          bisekcja lewy srodek
        else
          bisekcja (srodek + 1) prawy
  in bisekcja 0 num;;
```

3.5 Metoda Newtona liczenia pierwiastków

Kolejny element dziedziny algorytmicznej: liczby rzeczywiste i operacje arytmetyczne. Uwaga na kropki.

Specyfikacja $\text{sqrt } x$:

- warunek początkowy: zakładamy, że $x \geq 0$,
- warunek końcowy: chcemy mieć procedurę, której wynikiem jest przybliżenie pierwiastka zadaną dokładnością ε : $|\text{sqrt } x - \sqrt{x}| \leq \varepsilon$.

Metoda: zaczynamy od pewnej wartości i iteracyjnie poprawiamy ją, aż wynik będzie dobry.

```
let epsilon = ...;;
let sqrt x =
  let rec sqrt_iter g =
    if good g x then
      g
    else
      sqrt_iter (improve g x)
  in
    sqrt_iter (guess x);;
```

Predykat good decyduje o tym, kiedy przybliżenie jest dobre. Wystarczy więc, że przyjmiemy następującą specyfikację $\text{good } g\ x$, a będziemy mieli zapewnioną częściową poprawność algorytmu:

- warunek początkowy: $x \geq 0, g \geq 0$,
- $\text{good } g\ x \implies |g - \sqrt{x}| \leq \varepsilon$

Oznaczmy $e = g - \sqrt{x}$. Wówczas mamy:

$$\begin{aligned} |g^2 - x| &= |g^2 - (\sqrt{x})^2| = |(g - \sqrt{x})(g + \sqrt{x})| = \\ &= |e|(g + \sqrt{x}) \end{aligned}$$

Stąd:

$$|e| = \frac{|g^2 - x|}{g + \sqrt{x}}$$

Czyli:

$$|e| \leq \varepsilon \Leftrightarrow \frac{|g^2 - x|}{g + \sqrt{x}} \leq \varepsilon \Leftrightarrow |g^2 - x| \leq \varepsilon (g + \sqrt{x}) \Leftrightarrow |g^2 - x| \leq \varepsilon \cdot g$$

Tak więc, możemy zaimplementować predykat `good` w następujący sposób:

```
let square x = x *. x;;  
let good g x = abs_float ((square g) -. x) <= epsilon *. g;;
```

Zauważmy, że bez względu na to, jak zostaną zaimplementowane procedury `guess` i `improve`, mamy już zapewnioną częściową poprawność procedury `sqrt`. Zapewnienie własności stopu nie będzie już takie proste.

Zastanówmy się przez chwilę, czego oczekujemy od procedur `guess` i `improve`? Chcemy, aby nasza iteracja kiedyś się zakończyła, czyli:

$$\exists_{n \in \mathbb{N}} \text{good}((\text{fun } g \rightarrow \text{improve } g \ x)^n \ g)$$

Jeżeli przybliżenie g nie jest dostatecznie dobre (oraz $g > 0$), to szukany pierwiastek leży gdzieś między g , a $\frac{x}{g}$. Dobrym kandydatem jest więc średnia arytmetyczna tych liczb:

```
let average x y = (x +. y) /. 2.0;;  
let improve g x = average g (x /. g);;
```

Warunek początkowy tak zaimplementowanej procedury `improve`, to:

$$g > 0, \ x \geq 0$$

Równocześnie, dla takich danych, spełniony jest następujący warunek końcowy:

$$\text{improve } g \ x > 0$$

Podany warunek początkowy `improve` wymaga, aby pierwsze przybliżenie było dodatnie: `guess x > 0`. Łatwo jednak temu sprostać, wystarczy za pierwsze przybliżenie przyjąć np. 1:

```
let guess x = 1.0;;
```

Zweryfikujemy teraz własność stopu naszego algorytmu. Zauważmy, że kolejne przybliżenia pierwiastka są dodatnie. Zaczynamy od 1, a każdy kolejny krok przekształca dodatnie przybliżenie w dodatnie. Błąd kolejnego przybliżenia wynosi:

$$\begin{aligned} \frac{g + \frac{x}{g}}{2} - \sqrt{x} &= \frac{\sqrt{x} + e + \frac{x}{\sqrt{x+e}}}{2} - \sqrt{x} = \frac{\sqrt{x} + e + \frac{x}{\sqrt{x+e}} - 2\sqrt{x}}{2} = \\ &= \frac{e - \sqrt{x} + \frac{x}{\sqrt{x+e}}}{2} = \frac{(e - \sqrt{x})(e + \sqrt{x}) + x}{2(\sqrt{x} + e)} = \\ &= \frac{e^2 - x + x}{2(\sqrt{x} + e)} = \frac{e^2}{2(\sqrt{x} + e)} = \\ &= \frac{e}{2} \cdot \frac{1}{1 + \frac{\sqrt{x}}{e}} \end{aligned}$$

Rozważmy przypadki:

- Jeśli błąd jest ujemny, to $-\sqrt{x} < e < 0$, stąd $\frac{\sqrt{x}}{e} < -1$, czyli $1 + \frac{\sqrt{x}}{e} < 0$. Tak więc, w następnym kroku błąd jest dodatni.
- Jeśli błąd jest dodatni, to zauważmy, że $\frac{1}{1 + \frac{\sqrt{x}}{e}} < 1$, a więc błąd kolejnego przybliżenia jest przynajmniej dwukrotnie mniejszy.
- Jeśli błąd jest równy zero, to wychodzimy oczywiście z pętli.

Stąd, wszystkie przybliżenia są dodatnie, a tylko w pierwszym kroku błąd może być ujemny. Tak więc, jako funkcję miary możemy przyjąć:

$$\mu(g) = \begin{cases} \max\left(\left\lceil \log_2 \frac{2 \cdot (g - \sqrt{x})}{\varepsilon} \right\rceil, 0\right) & \text{dla } g > \sqrt{x} \\ 1 + \mu\left(\frac{g + \sqrt{x}}{2}\right) & \text{dla } g < \sqrt{x} \\ 0 & \text{dla } g = \sqrt{x} \end{cases}$$

Jeżeli wartość funkcji miary jest równa zero, to błąd przybliżenia jest nieujemny oraz:

$$g - \sqrt{x} \leq \frac{\varepsilon}{2}$$

Stąd:

$$|g^2 - x| = (g - \sqrt{x})(g + \sqrt{x}) \leq \frac{\varepsilon}{2} \cdot (g + \sqrt{x}) \leq \frac{\varepsilon}{2} \cdot 2g = \varepsilon \cdot g$$

Czyli:

$$|g^2 - x| \leq \varepsilon \cdot g \implies \text{good } g \ x$$

Jak widać, w przypadku tego algorytmu, dowód własności stopu jest trudniejszy niż dowód poprawności.

3.6 Podział problemu na podproblemy i technika pobożnych życzeń

Jak rozbiliśmy problem na podproblemy:

- iterowanie przybliżeń, aż są dobre: `sqrt`, `sqrt_iter`,
- pierwsze przybliżenie: `guess`,
- kiedy przybliżenie jest dobre: `good`,
- poprawienie przybliżenia: `improve`,
- podnoszenie do kwadratu: `square`,
- średnia: `average`,
- wartość bezwzględna: `abs`.

Jak dzielić problem na podproblemy? To trudna umiejętność, która istotnie wpływa na to czy będziemy dobrymi informatykami i czy nasze programy będą czytelne. Należy się tu kierować zasadami *dekompozycji* i *abstrakcji proceduralnej*:

- Należy podzielić problem trudniejszy na kilka prostszych „zgodnie z jego strukturą”, tzn. tak, aby można było prosto wyrazić rozwiązanie całego problemu za pomocą rozwiązań podproblemów — w ten sposób powstające definicje są proste i łatwe do ogarnięcia.

- Ponadto należy tak formułować podproblemy, aby były jak najbardziej ogólne — wówczas w większych systemach istnieje możliwość wykorzystania jednego rozwiązania w kilku miejscach.

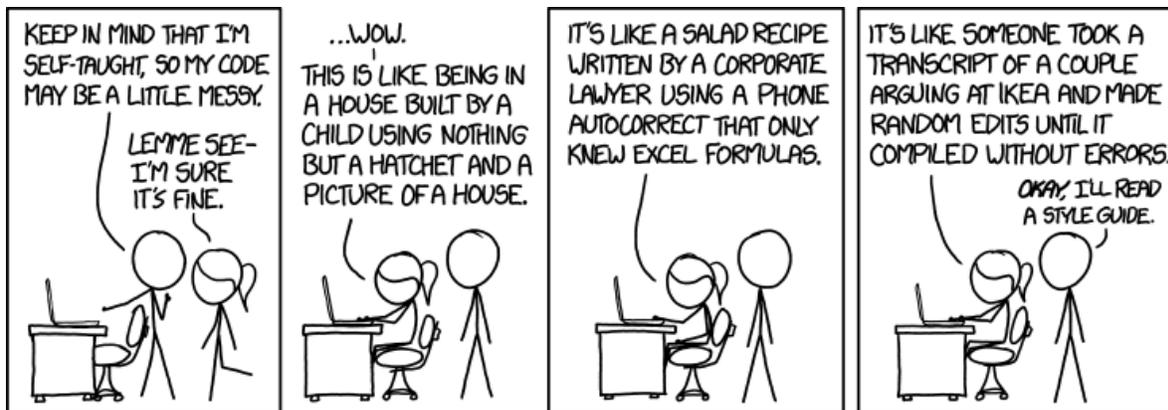
Zasada pobożnych życzeń:

- Tworząc rozwiązanie problemu powinniśmy rozbić go na podproblemy i zapisać rozwiązanie zakładając na chwilę, że podproblemy są już rozwiązane, a dopiero potem приступujemy do ich rozwiązywania. Jest to tzw. zasada *pobożnych życzeń*. W ten sposób rozwiązujemy problem *top-down* — od ogółu do szczegółów.

Kilka innych dobrych zasad:

- Black-box approach: Polegamy na tym, że dana wartość/procedura spełnia specyfikację, ale abstrahujemy od sposobu jej zaimplementowania. Podprocedury traktujemy chwilowo jak abstrakcje wszystkich możliwych rozwiązań tych podproblemów.
- Information hiding: Użytkownika naszego rozwiązania problemu nie interesuje jakie podprocedury mamy i co one robią. Zastosowanie definicji lokalnych do ukrycia sposobu rozwiązania. Dzięki temu nie zakładamy nic o sposobie rozwiązania problemu i korzystamy z niego w sposób bardziej elastyczny. Jeśli w przyszłości zechcemy zmienić takie rozwiązanie, to zrobimy to bezboleśnie.
- Separation of concerns: Dzięki ukrywaniu informacji o sposobie rozwiązania różnych problemów, są one od siebie niezależne i nie kolidują ze sobą. Na raz, możemy zajmować się tylko jednym problemem.

3.7 Deser



<http://xkcd.com/1513/>

Ćwiczenia

Rozwiązania poniższych zadań to proste procedury operujące na liczbach całkowitych. Ich rozwiązanie wymaga zastosowania rekurencji ogonowej. Dla każdej procedury rekurencyjnej podaj jej specyfikację, tj. warunek początkowy i końcowy, oraz uzasadnij jej poprawność. Poprawność procedur rekurencyjnych można pokazać przez indukcję. Nie zapomnij o uzasadnieniu własności stopu.

1. Udowodnij, że dla każdego naturalnego n zachodzi $\text{fib } n = \text{Fib}_n$. Podaj specyfikację dla `fib_pom` i udowodnij ją przez indukcję.

```
let fib n =  
  let rec fib_pom a b n =  
    if n = 0 then a else fib_pom b (a + b) (n - 1)  
  in  
    fib_pom 0 1 n;;
```

2. Potęgowanie liczb całkowitych (z akumulatorem).
3. [Bentley] Dana jest funkcja $f : \{1, 2, \dots, n\} \rightarrow \mathcal{Z}$. Należy znaleźć takie $1 \leq a \leq b \leq n$, że suma $\sum_{i=a}^b f(i)$ jest największa. Napisz taką procedurę `p`, że `p f n = b - a + 1` (dla a i b jak wyżej).
4. Dana jest funkcja nierosnąca $f : \text{int} \rightarrow \text{int}$, taka, że $f(x + 1) \geq f(x) - 1$. Napisz procedurę, która znajduje punkt stały tej funkcji, tzn. taką liczbę x , że $f(x) = x$ (lub jeśli punkt stały nie istnieje, to taką liczbę x , że $f(x - 1) \geq x$ oraz $f(x) \leq x$).
5. Napisz procedurę `zera_silni : int → int`, która dla danej dodatniej liczby całkowitej n obliczy ile zer znajduje się na końcu zapisu dziesiętnego liczby $n!$.
6. Dana jest różnowartościowa funkcja $f : \text{int} \rightarrow \text{int}$. Napisz procedurę `maksima`, która dla danych liczb l i p ($l \leq p$) obliczy ile lokalnych maksimumów ma funkcja f w przedziale $[l, p]$.
7. Przypomnij sobie rozwiązania zadań z poprzedniego wykładu. Jeśli rozwiązałeś je nie stosując rekurencji ogonowej, to czy potrafisz je rozwiązać (lepiej) stosując ją?

Wytyczne dla prowadzących ćwiczenia

W każdym zadaniu należy zastosować rekurencję ogonową. Przy rozwiązywaniu zadań kładziemy nacisk na poprawność, a także niejawnie na efektywność. W szczególności każda procedura powinna być wyspecyfikowana i zweryfikowana. W przypadku procedur rekurencyjnych pokazujemy również własność stopu. Należy zwrócić uwagę na niezmienniki.

Uwagi do zadań:

- Ad. 2** To zadanie należy rozwiązać w dwóch wersjach: najpierw prościej (liniowa złożoność czasowa), a następnie bardziej efektywnie (logarytmiczna złożoność czasowa). W obu przypadkach ogonowo.
- Ad. 3** To zadanie można rozwiązać w liniowej złożoności czasowej. Jeżeli studenci nie uważają tego od razu, należy najpierw rozwiązać je mniej efektywnie (i w bardziej skomplikowany sposób).
- Ad. 4** Efektywne rozwiązanie tego zadania jest bardziej skomplikowane niż proste nieefektywne rozwiązanie. Można zrobić oba warianty.
- Ad. 7** „Lepiej”, to bardziej efektywnie.

Wykład 4. Struktury danych

Jak dotąd, definiowane przez nas pojęcia to wyłącznie stałe kilku wbudowanych typów i procedury. O ile możemy definiować złożone procedury, to nie jesteśmy w stanie definiować złożonych danych. Zajmiemy się tym na wykładzie. Zajmiemy się również metodyką tworzenia nowych *typów* danych.

Pojęcie typu danych jest zapewne dobrze znane Czytelnikowi z innych języków programowania. Spróbujmy jednak, dla porządku, krótko scharakteryzować to pojęcie. Typ danych to zbiór wartości wraz z zestawem podstawowych operacji na tych wartościach.

W każdym języku programowania dostępna jest pewna liczba wbudowanych typów danych, takich jak liczby całkowite, wartości logiczne itp. Są one nazywane typami *prostymi*. Typy proste wbudowane w dany język programowania tworzą dziedzinę algorytmiczną. Oprócz typów prostych mamy typy *złożone*, które zawierają w sobie prostsze typy. W tym wykładzie poznamy różne konstrukcje pozwalające na tworzenie rozmaitych typów złożonych.

4.1 Typy wbudowane

Poznaliśmy już kilka podstawowych typów danych wbudowanych w język Ocaml wraz z operacjami na nich. Są to: `bool`, `int`, `float`, `char` i `string`.

W Ocamlu środowisko przyporządkowuje każdej nazwie stałej jej wartość, która jest określonego typu. Dotyczy to również nazwanych procedur i w ich przypadku oznacza, że wiadomo jakiego typu są argumenty i wynik. W szczególności operacje arytmetyczne na liczbach całkowitych i zmiennopozycyjnych muszą mieć różne nazwy, gdyż ich argumenty są różnych typów. Operacje arytmetyczne na liczbach całkowitych zapisujemy w klasyczny sposób: `+`, `*`, itd., a operacje na liczbach zmiennopozycyjnych mają dodaną w nazwie kropkę: `+.` , `*.` , Niestety, zapewne na początku często będą się zdarzać Czytelnikom błędy wynikające z mylenia tych operacji.

4.2 Konstruktory

Konstruktory to szczególnego rodzaju operacje tworzące wartości określonych typów złożonych. Jednak konstruktor to nie tylko procedura tworząca złożoną wartość, to coś więcej. Dodatkowo są one odwracalne, tzn. z ich wyniku można jednoznacznie odtworzyć ich argumenty. Dzięki temu konstruktory nie tylko służą do tworzenia wartości typów złożonych, ale również do rozbijania ich na tworzące je wartości typów prostszych.

Formalnie, konstruktory (wartości określonego typu) mają następujące własności:

- są one różnowartościowe,
- różne konstruktory wartości tego samego typu mają różne wartości, tzn. przeciwdziedziny różnych konstruktorów są rozłączne,
- wszystkie konstruktory wartości danego typu są łącznie "na", tzn. każdą wartość danego typu można skonstruować,
- każdą wartość danego typu można przedstawić w postaci wyrażenia złożonego z konstruktorów i stałych innych typów.

Na razie nie znamy jeszcze żadnych konstruktorów. Zostaną one przedstawione dalej, wraz z kolejnymi typami złożonymi. Wówczas zobaczymy przykłady ich użycia.

4.3 Wzorce

Zanim przedstawimy sposoby konstruowania złożonych typów danych, wprowadzimy pojęcie *wzorca*. Co to jest wzorec? Wzorec jest to rodzaj „wyrażenia”. Do budowy wzorców możemy używać konstruktorów i identyfikatorów. Wzorce pojawiają się w takich kontekstach, że zawsze dopasowywana jest do nich wartość pewnego wyrażenia. Identyfikatory występujące we wzorcach są nazwami nowych stałych wprowadzanych do środowiska. Jeśli wartość wyrażenia dopasowywanego do wzorca "pasuje", to występującym we wzorcu identyfikatorom przypisywane są w środowisku wartości odpowiadających im składowych dopasowywanej wartości. Identyfikatory występujące we wzorcu nie mogą się powtarzać, gdyż każde wystąpienie identyfikatora odpowiada za inny fragment dopasowywanej wartości. Wzorce będą stanowiły podstawowy mechanizm dekompozycji wartości złożonych typów, jednak ich zastosowanie nie ogranicza się tylko do tego.

Przedstawiając kolejne typy złożone będziemy przedstawiać również towarzyszące im konstruktory i wzorce. Istnieje jednak kilka podstawowych wzorców, które mogą być używane dla wartości dowolnego typu:

$$\langle \text{wzorec} \rangle ::= \langle \text{identyfikator} \rangle \mid _ \mid \langle \text{stała} \rangle \mid \dots$$

- identyfikator — pasuje do każdej wartości, nazwie przyporządkowywana jest dana wartość,
- `_` — tak jak identyfikator, tylko nie wprowadza żadnej nowej nazwy,
- stała (np. liczbowa) — pasuje tylko do samej siebie.

Wzorca `_` używamy wówczas, gdy interesuje nas sam fakt, że wartość pasuje do wzorca, a nie chcemy wprowadzać do środowiska nowych nazw.

Wzorców możemy używać w konstrukcji `match-with` oraz w definicjach procedur, zarówno procedur nazwanych, jak i λ -abstrakcji:

$$\begin{aligned} \langle \text{definicja} \rangle &::= \underline{\text{let}} \langle \text{wzorec} \rangle \underline{=} \langle \text{wyrażenie} \rangle \mid \\ &\underline{\text{let}} [\underline{\text{rec}}] \langle \text{identyfikator} \rangle \{ \langle \text{wzorec} \rangle \}^* \underline{=} \langle \text{wyrażenie} \rangle \\ \langle \text{wyrażenie} \rangle &::= \underline{\text{match}} \langle \text{wyrażenie} \rangle \underline{\text{with}} \{ \langle \text{wzorec} \rangle \underline{\rightarrow} \langle \text{wyrażenie} \rangle \mid \}^* \\ &\langle \text{wzorec} \rangle \underline{\rightarrow} \langle \text{wyrażenie} \rangle \mid \\ &\underline{\text{function}} \{ \langle \text{wzorec} \rangle \underline{\rightarrow} \langle \text{wyrażenie} \rangle \mid \}^* \\ &\langle \text{wzorec} \rangle \underline{\rightarrow} \langle \text{wyrażenie} \rangle \end{aligned}$$

Najprostsze zastosowanie wzorców to definicje stałych. Po lewej stronie `=` możemy mieć zamiast identyfikatora wzorec. Wartość wyrażenia po prawej stronie `=` jest dopasowywana do wzorca, a wszystkie identyfikatory, którym w ten sposób są przypisywane wartości, trafiają do środowiska.

Przykład:

```
3;;
- : int = 3

let _ = 3;;
```

```
- : int = 3

let a = 42;;
val a : int = 42
```

W definicjach procedur nazwa procedury musi być identyfikatorem, ale parametry formalne mogą być wzorcami. Wówczas, w momencie wywołania procedury wartości argumentów są dopasowywane do parametrów formalnych. Wszystkie identyfikatory, którym w ten sposób są przypisywane wartości, trafiają do tymczasowego środowiska powstającego dla obliczenia wartości procedury.

Przykład:

```
let f 6 9 = 42;;
val f : int ->int ->int

f 6 9;;
- : int = 42

let p x _ = x + 1;;
val p : int ->'a ->int = <fun>

p 6 9;;
- : int = 7
```

W wyrażeniach `match-with` wartość wyrażenia występującego po `match` jest dopasowywana do kolejnych wzorców. Wzorzec, do którego ta wartość pasuje, wskazuje równocześnie wyrażenie, którego wartość jest wartością całego wyrażenia `match-with`. Jeśli dopasowywana wartość pasuje do kilku wzorców, to wybierany jest pierwszy z nich. Jeżeli nie pasuje ona do żadnego z wzorców, to zgłaszany jest błąd (wyjątek).

Podobnie działa dopasowywanie wzorców w λ -abstrakcjach. Argument procedury jest dopasowywany do kolejnych wzorców. Wzorzec, do którego wartość argumentu pasuje wskazuje równocześnie wyrażenie, którego wartość jest wynikiem procedury. Jeśli wartość argumentu pasuje do kilku wzorców, to wybierany jest pierwszy z nich. Jeżeli nie pasuje ona do żadnego z wzorców, to zgłaszany jest błąd (wyjątek).

Przykład:

```
let rec silnia =
  function 0 ->1 | x ->x * silnia (x - 1);;

let rec silnia n =
  match n with
  0 ->1 |
  x ->x * silnia (x-1);;
```

Konstrukcja `match-with` jest tylko lukrem syntaktycznym rozwijanym do zastosowania odpowiedniej λ -abstrakcji.

Przykład: Konstrukcja `match-with` z poprzedniego przykładu jest rozwijana w następujący sposób:

```
let rec silnia n =
  (function
    0 ->1 |
    x ->x * silnia (x-1)
  ) n;;
```

4.4 Produkty kartezjańskie

Produkt kartezjański jako typ jest dokładnie tym, czym jest w matematyce. Jest to zbiór par, lub ogólniej n -ek wartości prostszych typów. Produkt kartezjański typów t_1, \dots, t_n jest oznaczany przez $t_1 * \dots * t_n$.

Każdemu typowi produktowemu towarzyszy jeden konstruktor postaci: (x_1, \dots, x_n) . Wartością takiego konstruktora jest n -ka podanych wartości.

Na wartościach produktów kartezjańskich jest określona równość. Dwie n -tki są równe, jeśli ich odpowiadające sobie współrzędne są równe.

$$\begin{aligned} \langle \text{wyrażenie} \rangle &::= \underline{([\langle \text{wyrażenie} \rangle \{ \underline{\quad}, \langle \text{wyrażenie} \rangle \}^*] \underline{\quad})} \\ \langle \text{wzorzec} \rangle &::= \underline{([\langle \text{wzorzec} \rangle \{ \underline{\quad}, \langle \text{wzorzec} \rangle \}^*] \underline{\quad})} \end{aligned}$$

Przykład:

```
(1,2,3);;
- : int * int * int = (1,2,3)

(42,(6.9,"ala"));
- : int * (float * string) = (42,(6.9,"ala"))

let (x,s) = (4.5,"ala");;
val x : float = 4.5
val s : string = "ala"

let fib n =
  let rec fib_pom n =
    if n = 0 then
      (0,1)
    else
      let (a,b) = fib_pom (n - 1)
      in (b,a + b)
  in
    let (a,b) = fib_pom n
    in a;;
val fib : int ->int = <fun>

let para x y = (x,y);;
```

```

val para : 'a ->'b ->'a * 'b = <fun>

let rzut_x (x,_) = x;;
val rzut_x : 'a * 'b ->'a = <fun>

let rzut_y (_,y) = y;;
val rzut_y : 'a * 'b ->'b = <fun>

```

Produkt kartezjański nie jest łączny, tzn. poniższe wyrażenia są różnych typów:

```

(2,3,4);;
- : int * int * int = (2,3,4)

(2,(3,4));;
- : int * (int * int) = (2,(3,4))

((2,3),4);;
- : (int * int) * int = ((2,3),4)

```

Jednoelementowe n -ki wartości typu t to, po prostu, wartości typu t , $(x) = x$. Istnieje „jedyneką” produktu kartezjańskiego (odpowiednik typu `void`), typ `unit`. Jedyłą wartością tego typu jest `()`. Jednak `t * unit` to inny typ niż `t`.

4.5 Listy

Listy to ciągi elementów tego samego typu (skończone lub nieskończone, ale od pewnego miejsca okresowe). Typ listy elementów typu t oznaczamy przez t `list`. Pierwszy element listy przyjęło się nazywać *głową*, a listę złożoną z wszystkich pozostałych elementów listy przyjęło się nazywać *ogonem*. Tak więc każda niepusta lista (tak jak wąż ;-)) składa się z głowy i ogona.

Typowi listowemu towarzyszą dwa konstruktory: `h::t i []`. `[]` to konstruktor listy pustej, a `h::t` tworzy listę, której głową jest `h`, a ogonem jest `t`.

Chcąc podać listę n -elementową nie musimy pisać: `x1 :: (x2 :: ... :: (xn :: []))...`, lecz możemy użyć skrótu notacyjnego: `[x1; x2; ...; xn]`. Dostępna jest też operacja sklejania list `@`.

Należy pamiętać, że operacje `::` i `[]` obliczane są w czasie stałym, natomiast `@` w czasie proporcjonalnym do długości pierwszego argumentu.

Na listach jest określona równość. Dwie listy są równe, jeżeli są równej długości i odpowiadające sobie elementy są równe.

Przykład:

```

[];;
- : 'a list = []

1::2::3::[];;
- : int list = [1; 2; 3]

[1; 2; 3; 4];;
- : int list = [1; 2; 3; 4]

```

```

[1;2;3] @ [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]

["To"; "ci"; "dopiero"; "lista"];;
- : string list = ["To"; "ci"; "dopiero"; "lista"]

let length l =
  let rec pom a l =
    match l with
    [] ->a |
    _::t ->pom (a+1) t
  in
  pom 0 l;;
val length : 'a list ->int = <fun>

let sumuj l =
  let rec pom a l =
    match l with
    [] ->a |
    h::t ->pom (a+h) t
  in
  pom 0 l;;
val sumuj : int list ->int = <fun>

let rec listapar2paralist =
  function
  [] ->([],[]) |
  (x,y)::t ->
    let (l1,l2) = listapar2paralist t
    in (x :: l1,y :: l2);;
val listapar2paralist : ('a * 'b) list ->'a list * 'b list = <fun>

```

Możliwe jest definiowanie list nieskończonych. Należy jednak zdawać sobie sprawę, że listy są implementowane jako wskaźnikowe listy jednokierunkowe. Dlatego też lista nieskończona może mieć co najwyżej postać cyklu z „ogonkiem”. Poniższy przykład pokazuje jak definiować listy nieskończone. Pokazuje on również, że nie tylko procedury mogą być obiektami definiowanymi rekurencyjnie.

Przykład: Definicje list nieskończonych:

```

let rec jedyнки = 1::jedyнки;;
val jedyнки : int list = [1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; ...]

let rec cykl = 1 :: 0 :: -1 :: 0 :: cykl;;
val cykl : int list = [1; 0; -1; 0; 1; 0; -1; 0; ...]

[1;2;3] @ cykl;;
- : int list = [1; 2; 3; 1; 0; -1; 0; 1; 0; -1; 0; ...]

```

```
cykl @ [1;2;3];;
```

Stack overflow during evaluation (looping recursion?).

Moduł `List` zawiera wiele poręcznych procedur operujących na listach. O modułach będziemy mówić dokładniej w jednym z wykładów. W tej chwili wystarczy nam wiedzieć, że aby korzystać z pojęć zdefiniowanych w module `List`, wystarczy wprowadzić polecenie `open List;;` lub przed każdą nazwą pojęcia dodać przedrostek postaci `List`. Moduł ten implementuje m.in. następujące operacje:

- `length` — oblicza długość listy (uwaga: działa w czasie proporcjonalnym do długości listy),
- `hd` — zwraca głowę (niepustej) listy,
- `tl` — zwraca ogon (niepustej) listy,
- `rev` — oblicza listę złożoną z tych samych elementów, ale w odwrotnej kolejności (uwaga: działa w czasie proporcjonalnym do długości listy),
- `nth` — zwraca element z podanej pozycji na liście (głowa ma numer 0, uwaga: działa w czasie proporcjonalnym do numeru pozycji plus 1),
- `append` — sklejanie list, działa tak jak `@`, ale nie jest zapisywane infiksowo, tylko tak, jak zwykła procedura.

Przykład: Pokazać przykładowe implementacje:

- Wyszukanie n -tego elementu listy:

```
let rec nth l n =
  if n = 0 then hd l
  else nth (tl l) (n - 1);;
```

- Odwrócenie listy `rev`:

```
let rev l =
  let rec pom l w =
    if l = [] then w
    else pom (tl l) ((hd l) :: w)
  in pom l [];;
```

- Sklejanie list `@`, `append`:

```
let rec append l1 l2 =
  if l1 = [] then l2
  else (hd l1) :: (append (tl l1) l2);;
```

Przykład:

```
open List;;
length ["To"; "ci"; "dopiero"; "lista"];;
- : int = 4

hd ["To"; "ci"; "dopiero"; "lista"];;
- : string = "To"

tl ["To"; "ci"; "dopiero"; "lista"];;
- : string list = ["ci"; "dopiero"; "lista"]

rev ["To"; "ci"; "dopiero"; "lista"];;
- : string list = ["lista"; "dopiero"; "ci"; "To"]

nth ["To"; "ci"; "dopiero"; "lista"] 2;;
- : string = "dopiero"

append [1; 2; 3] [4; 5];;
- : int list = [1; 2; 3; 4; 5]
```

4.6 Deklaracje typów

Zwykle kompilator Ocaml'a sam jest w stanie wywnioskować jakiego typu są wyrażenia. Istnieje jednak możliwość nazywania typów i określania jakiego typu się spodziewamy. Wówczas kompilator sprawdza, czy dane wyrażenie faktycznie jest takiego typu, jak chcieliśmy, a zamiast pełnego rozwinięcia typu może używać podanej przez nas krótszej nazwy. Niektóre typy danych można wprowadzić tylko poprzez ich deklarację. Są to te typy, których deklaracje wprowadzają nowe konstruktory.

Możliwe jest też zdefiniowanie typów sparametryzowanych, tzn. takich, w których typy niektórych ich składowych są podawane jako parametry typu. Elementy struktury danych określone jako parametry mogą być dowolnego typu, ale wszystkie elementy określone przez ten sam parametr muszą być (w danej wartości) tego samego typu. (Uważny czytelnik zapewne domyśla się, że przykładem typu sparametryzowanego jest lista. Parametrem typu listowego jest typ elementów listy.) Parametry typu podajemy zwyczajowo przed nazwą typu sparametryzowanego, np. `int list`.

```

⟨jednostka kompilacji⟩ ::= ⟨deklaracja typu⟩
⟨deklaracja typu⟩      ::= type ⟨identyfikator⟩ ≡ ⟨typ⟩ |
                           type ⟨parametr typowy⟩ ⟨identyfikator⟩ ≡ ⟨typ⟩ |
                           type ( ⟨parametr typowy⟩ {  _  ⟨parametr typowy⟩ } * )
                           ⟨identyfikator⟩ ≡ ⟨typ⟩
⟨typ⟩                  ::= ⟨identyfikator⟩ |
                           ⟨typ⟩ ⟨identyfikator⟩ |
                           ( ⟨typ⟩ {  _  ⟨typ⟩ } * ) ⟨identyfikator⟩ |
                           ⟨typ⟩ { * ⟨typ⟩ } * |
                           ( ⟨typ⟩ ) | ...
⟨parametr typowy⟩     ::=  _  ⟨identyfikator⟩

```

Przykład:

```

type p = int * float;;
type 'a lista = 'a list;;
type ('a,'b) para = 'a * 'b;;
type 'a t = ('a,'a) para * ('a list) list;;

```

Podstawowym zastosowaniem deklaracji typów (oprócz wprowadzania typów, które wymagają zadeklarowania) jest definiowanie typów złożonych, które występują w interfejsach procedur. Wówczas można podać, iż oczekujemy, że dany argument powinien być określonego typu. W ogólności można dla dowolnego wzorca lub wyrażenia podać typ, jakiego oczekujemy.

$$\begin{aligned}
\langle \text{wyrażenie} \rangle &::= \underline{(\langle \text{wyrażenie} \rangle \underline{ : } \langle \text{typ} \rangle \underline{)}} \\
\langle \text{wzorzec} \rangle &::= \underline{(\langle \text{wzorzec} \rangle \underline{ : } \langle \text{typ} \rangle \underline{)}}
\end{aligned}$$

Przykład: Specyfikacja typów.

```

type point = float * float;;
type point = float * float

type vector = point;;
type vector = point

let (p:point) = (2.0,3.0);;
val p : point = (2.,3.)

let shift ((x,y): point) ((xo,yo): vector) =

```

```

    ((x +. xo,y +. yo) : point);;
val shift : point ->vector ->point = <fun>

shift p p;;
- : point = (4.,6.)

```

4.7 Rekordy

Typy rekordowe są podobne do produktów kartezjańskich. Rekord odpowiada n -ce, ale współrzędne (tzw. pola rekordu) są identyfikowane raczej po nazwie, a nie wg pozycji w n -ce. Typy rekordowe występują w wielu językach programowania — w C i C++ są nazywane strukturami. Typy rekordowe wprowadza się deklarując typ postaci:

$$\langle \text{typ} \rangle ::= \underline{\{ \{ \langle \text{identyfikator} \rangle : \langle \text{typ} \rangle ; \}^* \langle \text{identyfikator} \rangle : \langle \text{typ} \rangle [;] \}} \underline{\}$$

Identyfikatory to nazwy pól rekordów. Konstruktor wartości rekordowych ma postać:

$$\begin{aligned} \langle \text{wyrażenie} \rangle &::= \underline{\{ \{ \langle \text{identyfikator} \rangle \equiv \langle \text{wyrażenie} \rangle ; \}^* \langle \text{identyfikator} \rangle \equiv \langle \text{wyrażenie} \rangle [;] \}} \underline{\} \\ \langle \text{wzorzec} \rangle &::= \underline{\{ \{ \langle \text{identyfikator} \rangle \equiv \langle \text{wzorzec} \rangle ; \}^* \langle \text{identyfikator} \rangle \equiv \langle \text{wzorzec} \rangle [;] \}} \underline{\} \end{aligned}$$

Używając takiego konstruktora do budowy rekordu musimy podać wartości wszystkich pól (w dowolnej kolejności). Natomiast używając go jako wzorca, możemy podać tylko interesujące nas pola. Do pojedynczych pól rekordów możemy się również odwoływać tak, jak w innych językach programowania, podając nazwę pola po kropce.

$$\langle \text{wyrażenie} \rangle ::= \langle \text{wyrażenie} \rangle . \langle \text{identyfikator} \rangle$$

Przykład:

```

type ulamek = { licznik : int ; mianownik : int };;
type ulamek = { licznik : int; mianownik : int; }

let q = { licznik = 3; mianownik = 4 };;
val q : ulamek = {licznik = 3; mianownik = 4}

let { licznik = a ; mianownik = b } = q;;
val a : int = 3
val b : int = 4

let { licznik = c } = q;;
val c : int = 3

q.licznik;;
- : int = 3

```

Uwaga: Ze względu na przysłanianie nazw, należy unikać sytuacji, gdy dwa pola różnych typów rekordowych mają takie same nazwy. Wówczas będziemy mogli korzystać tylko z tego, które zostało zdefiniowane jako ostatnie.

4.8 Typy wariantowe

W Ocamlu istnieje również odpowiednik unii – typy wariantowe, zwane też typami algebraicznymi. Deklarujemy je w następujący sposób:

$$\begin{aligned} \langle \text{typ} \rangle & ::= \langle \text{wariant} \rangle \{ _ | \langle \text{wariant} \rangle \}^* \\ \langle \text{wariant} \rangle & ::= \langle \text{Identyfikator} \rangle [\text{of} \langle \text{typ} \rangle] \end{aligned}$$

Przez $\langle \text{Identyfikator} \rangle$ oznaczamy identyfikatory rozpoczynające się wielką literą. Natomiast $\langle \text{identyfikator} \rangle$ oznacza identyfikatory rozpoczynające się małą literą. Jak zobaczymy za chwilę, rozróżnienie to jest konieczne dla odróżnienia we wzorcach nazw konstruktorów od nazw wprowadzanych stałych.

Każdy zadeklarowany wariant wprowadza konstruktor o podanej nazwie. Formalnie konstruktor taki może mieć co najwyżej jeden argument, ale zawsze może to być argument odpowiedniego typu produktowego. Konstruktorów typów wariantowych używamy w następujący sposób:

$$\begin{aligned} \langle \text{wyrażenie} \rangle & ::= \langle \text{Identyfikator} \rangle [\langle \text{wyrażenie} \rangle] \\ \langle \text{wzorzec} \rangle & ::= \langle \text{Identyfikator} \rangle [\langle \text{wzorzec} \rangle] \end{aligned}$$

Na typach wariantowych jest określona równość. Dwie wartości są równe, jeśli są wynikiem tego samego konstruktora, a jeżeli jest to konstruktor sparametryzowany, to dodatkowo parametry konstruktora muszą być sobie równe.

Przykład: Typy wariantowe:

```
type znak = Dodatni | Ujemny | Zero;;
type znak = Dodatni | Ujemny | Zero
```

```
let nieujemny x =
  match x with
  | Ujemny ->false |
  | _ ->true;;
val nieujemny : znak ->bool = <fun>
```

```
let ujemny x =
  x = Ujemny;;
val ujemny : znak ->bool = <fun>
```

```
type zespolone =
  Prostokatny of float * float |
  Biegunowy of float * float ;;
type zespolone = Prostokatny of float * float | Biegunowy of float * float
```

```
let modul =
  function
  | Prostokatny (x,y) ->sqrt (square x +. square y) |
  | Biegunowy (r,_) ->r;;
val modul : zespolone ->float = <fun>
```

Deklaracje typów wariantowych mogą być rekurencyjne, co pozwala na konstruowanie drzew. Przypomina to zdefiniowanie w Pascalu typu wskaźnikowego do typu, który jest definiowany dalej i w którym można korzystać z danego typu wskaźnikowego.

Przykład:

```
type drzewo =  
  Puste |  
  Wezel of int * drzewo * drzewo;;
```

Podobnie jak w przypadku list, możemy tworzyć „nieskończone” drzewa poprzez ich zacyklenie.

Przykład: Zacyklone, nieskończone drzewa:

```
let rec t = Wezel (42,t,Puste);;  
val t : drzewo = Wezel (42,Wezel (42,(...),Puste),Puste)
```

Deklaracje typów wariantowych mogą być sparametryzowane. Jest to szczególnie przydatne wówczas, gdy nie chcemy do końca określać typu składowych elementów.

Przykład:

```
type 'a lista = Pusta | Pelna of 'a * 'a lista;;  
type 'a lista = Pusta | Pelna of 'a * 'a lista  
  
Pelna (4,Pusta);;  
- : int lista = Pelna (4,Pusta)  
  
Pelna ("ala",Pelna("ula",Pusta));;  
- : string lista = Pelna ("ala",Pelna ("ula",  
Pusta))  
  
Pelna (4,Pelna ("ula",Pusta));;  
error ...
```

4.9 Aliasy we wzorcach

Czasami, gdy używamy złożonych wzorców, chcemy równocześnie uchwycić złożoną wartość, jak i jej elementy. Możemy to zrobić za pomocą konstrukcji `as`.

$$\langle \text{wzorzec} \rangle ::= \langle \text{wzorzec} \rangle \text{ as } \langle \text{identyfikator} \rangle$$

Dopasowywana wartość jest przypisywana identyfikatorowi po prawej stronie `as` oraz jest dopasowywana do wzorca po lewej stronie.

```

let (x,y) as para = (2,3);;
val x : int = 2
val y : int = 3
val para : int * int = (2,3)

let (h::t) as lista = [1; 2; 3; 4];;
val h : int = 1
val t : int list = [2; 3; 4]
val lista : int list = [1; 2; 3; 4]

```

4.10 Wyjątki

Czasami chcemy zaprogramować w postaci procedur funkcje częściowe. W jaki sposób możemy to zrobić? Jaki powinien być wynik dla punktów spoza dziedziny? Możemy użyć tu mechanizmu *wyjątków*. W języku istnieje specjalny typ wariantowy `exn`. Jest to nietypowy typ, gdyż można na bieżąco rozszerzać zestaw wariantów tworzących ten typ. Wartości tego typu niosą informację o wyjątkowych sytuacjach uniemożliwiających wykonanie obliczeń. Nowe warianty typu `exn` możemy deklarować w następujący sposób:

$$\begin{aligned}
\langle \text{jednostka kompilacji} \rangle &::= \langle \text{deklaracja wyjątku} \rangle | \dots \\
\langle \text{deklaracja wyjątku} \rangle &::= \underline{\text{exception}} \langle \text{wariant} \rangle
\end{aligned}$$

Z wyjątkami można robić dwie podstawowe rzeczy: podnosić i przechwytywać. Podniesienie wyjątku, to wyrażenie, którego obliczenie „nie udaje się”, a porażce tej towarzyszy wartość podnieszonego wyjątku (typu `exn`). Jeśli obliczenie dowolnego podwyrażenia nie udaje się na skutek podniesienia wyjątku, to tak samo nie udaje się obliczenie całego wyrażenia. Tak więc wyjątek, to rodzaj propagującego się „błędu w obliczeniach”. Podniesienie wyjątku ma następującą postać, przy czym argument operacji `raise` musi być typu `exn`:

$$\langle \text{wyrażenie} \rangle ::= \underline{\text{raise}} \langle \text{wyrażenie} \rangle$$

Przechwycenie wyjątku to konstrukcja odwrotna do jego podniesienia. Możemy spróbować obliczyć wartość danego wyrażenia, licząc się z możliwością podniesienia wyjątku. Jeśli w danym wyrażeniu zostanie podniesiony wyjątek, to możemy go przechwycić i podać, jaka powinna być wartość całego wyrażenia w takim przypadku.

$$\langle \text{wyrażenie} \rangle ::= \underline{\text{try}} \langle \text{wyrażenie} \rangle \underline{\text{with}} \{ \langle \text{wzorzec} \rangle \underline{\text{->}} \langle \text{wyrażenie} \rangle \mid \}^* \langle \text{wzorzec} \rangle \underline{\text{->}} \langle \text{wyrażenie} \rangle$$

Przykład:

```

exception Dzielenie_przez_zero of float;;
exception Dzielenie_przez_zero of float

let dziel x y =
  if y = 0.0 then
    raise (Dzielenie_przez_zero x)

```

```

    else x /. y;;
val dziel : float ->float ->float = <fun>

let odwrotnosc x =
  try
    dziel 1.0 x
  with
    Dzielenie_przez_zero _ ->0.0;;
val odwrotnosc : float ->float = <fun>

odwrotnosc 42.;;
- : float = 0.0238095238095238082

odwrotnosc 0.0;;
- : float = 0.

```

Standardowo zdefiniowany jest wyjątek `Failure of string` oraz procedura:

```

let failwith s =
  raise (Failure s);;

```

Pamiętajmy, że należy odróżniać wartości typu `exn` od podniesienia wyjątku.

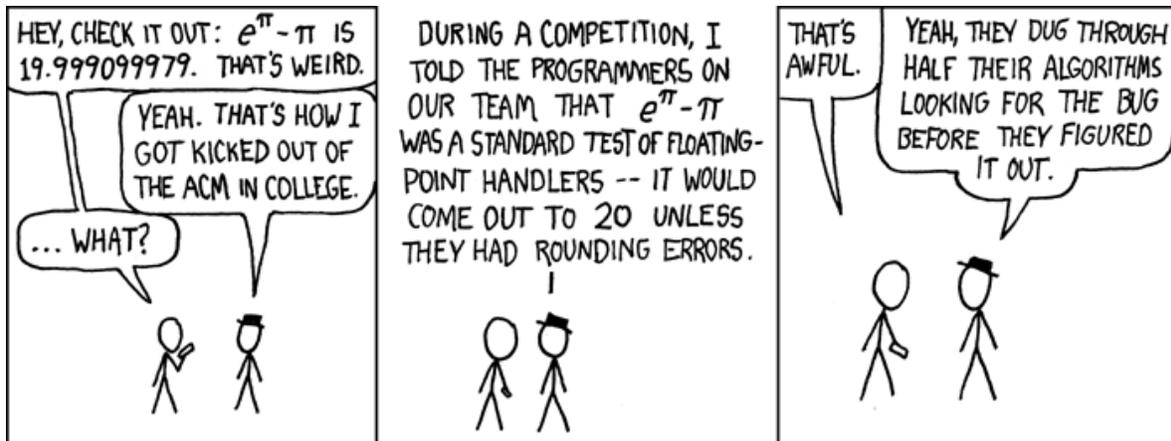
```

Dzielenie_przez_zero 4.5;;
- : exn = Dzielenie_przez_zero 4.5
raise (Dzielenie_przez_zero 4.5);;
Exception: Dzielenie_przez_zero 4.5.

```

Uwaga: W niektórych źródłach wyjątki są opisane jako konstrukcja imperatywna. Proszę jednak zwrócić uwagę, że nie odwołują się one do takich pojęć, jak zmienne, obiekty, stany, czy przypisanie. Dlatego też wyjątki są tu przedstawione jako mechanizm funkcyjny.

4.11 Deser



<http://xkcd.com/217/>

Ćwiczenia

Ćwiczenie programistyczne na listy i inne struktury danych:

1. Lista n pierwszych liczb naturalnych.
2. Zdublować elementy listy.
3. Napisz procedurę `last`, której wynikiem jest ostatni element listy.
4. Napisz procedury `head` i `tail`, które dla zadanej listy l i liczby całkowitej n zwracają pierwsze/ostatnie n elementów listy l . Jeśli lista l ma mniej elementów niż n , to wynikiem powinna być cała lista l . (Nazwy pochodzą od programów `head` i `tail` w Uniksie.) Co jest wynikiem `tail (head 1 n) 1`?
5. Napisz procedurę `shuffle : α list \rightarrow α list \rightarrow α list`, która dla danych dwóch list postaci $[x_1; x_2; \dots; x_n]$ oraz $[y_1; y_2; \dots; y_m]$ wyznaczy listę postaci $[x_1; y_1; x_2; y_2; \dots]$. Jeżeli jedna z list jest dłuższa, to jej końcowe elementy trafiają na koniec listy wykoowej. Na przykład: `shuffle [3; 2; 8; 1; 9; 3; 6] [5; 7; 0] = [3; 5; 2; 7; 8; 0; 1; 9; 3; 6]`.
6. Napisz procedurę, która listę par postaci $[(n_1, x_1); (n_2, x_2); \dots; (n_k, x_k)]$ przekształca w listę postaci $\underbrace{[x_1; \dots; x_1]}_{n_1 \text{ razy}}; \underbrace{[x_2; \dots; x_2]}_{n_2 \text{ razy}}; \dots; \underbrace{[x_k; \dots; x_k]}_{n_k \text{ razy}}$.
7. Załóżmy, że dana jest lista $[x_1; x_2; \dots; x_n]$. *Sufiksem* tej listy nazwiemy każdą listę, którą można uzyskać przez usunięcie pewnej liczby (od 0 do n) jej początkowych elementów. Tak więc sufiksami danej listy będzie n.p. ona sama, pusta lista, a także $[x_3; x_4; \dots; x_n]$. Napisz procedurę `tails : α list \rightarrow α list list`, która dla danej listy tworzy listę wszystkich jej sufiksów, uporządkowaną wg malejących ich długości.
8. Dane są dwie listy liczb całkowitych uporządkowane niemalejąco. Oblicz ile różnych liczb występuje na tych listach.
9. Napisz procedurę `zsumuj`, która dla danej niemalejącej listy dodatnich liczb całkowitych $(a_1 \dots a_n)$ oblicza listę $(b_1 \dots b_n)$, gdzie $b_i = \sum_{k=a_i}^n a_k$. (Pamiętaj o tym, że jeśli $m > n$, $\sum_{k=m}^n a_k = 0$.)
10. Napisz program wybierający `max` i `min` z listy i dokonujący jak najmniejszej liczby porównań.
11. Napisz program wybierający element dominujący z listy większościowej; uzasadnij jego poprawność.
12. Napisz procedurę `max_diff : int list \rightarrow int`, która dla niepustej listy $[x_1; \dots; x_n]$ znajdzie maksymalną różnicę $x_j - x_i$ dla $1 \leq i < j \leq n$.
13. Napisz procedurę `podziel : int list \rightarrow int list list`, która dla danej listy postaci $[a_1; a_2; \dots; a_n]$, zawierającej permutację zbioru $\{1, 2, \dots, n\}$ znajdzie jej podział na jak najliczniejszą listę list postaci:

$$[[a_1; a_2; \dots; a_{k_1}]; [a_{k_1+1}; a_{k_1+2}; \dots; a_{k_2}]; \dots; [a_{k_{m-1}+1}; a_{k_{m-1}+2}; \dots; a_{k_m}]]$$

taką że:

$$\begin{aligned}\{a_1, a_2, \dots, a_{k_1}\} &= \{1, 2, \dots, k_1\} \quad (\text{równość zbiorów}), \\ \{a_{k_1+1}, a_{k_1+2}, \dots, a_{k_2}\} &= \{k_1 + 1, k_1 + 2, \dots, k_2\}, \\ &\vdots \\ \{a_{k_{m-1}+1}, a_{k_{m-1}+2}, \dots, a_{k_m}\} &= \{k_{m-1} + 1, k_{m-1} + 2, \dots, k_m\}.\end{aligned}$$

Przyjmujemy, że wynikiem dla listy pustej jest lista pusta.

Przykład: `podziel [2; 3; 1; 6; 5; 4; 7; 9; 10; 11; 8] = [[2; 3; 1]; [6; 5; 4]; [7]; [9; 10; 11; 8]]`.

14. Napisz procedurę `mieszaj`, która wywołana dla danej listy, obliczy listę powstałą przez następujące przestawienie elementów: na pozycjach nieparzystych mają kolejno znajdować się $\lceil \frac{n}{2} \rceil$ początkowe elementy danej listy, a na pozycjach parzystych mają znajdować się pozostałe elementy danej listy, w odwrotnej kolejności. Na przykład: `mieszaj [1; 2; 3; 4; 5] = [1; 5; 2; 4; 3]`
15. Napisz procedurę `trójki: int list → (int * int * int) list`, która dla zadanej listy dodatnich liczb całkowitych, uporządkowanej rosnąco, stworzy listę takich trójek (a, b, c) liczb z danej listy, że:

- $a < b < c$,
- liczby a, b i c spełniają nierówność trójkąta, czyli $c < a + b$.

16. Dana jest lista liczb całkowitych $l = [x_1; \dots; x_n]$ uporządkowana niemalejąco. Napisz procedurę:

`malo: int list → int`

która dla danej listy l oblicza:

$$\min \{|x_i + x_j| : 1 \leq i < j \leq n\}$$

Na przykład, `malo [-42, -12, -8, -1, -1, 5, 15, 60] = 2`.

Możesz założyć, że dana lista l zawiera przynajmniej dwa elementy.

17. Dana jest niepusta, rosnąca lista liczb całkowitych $l = [x_1; \dots; x_n]$. Napisz procedurę:

`fit: int → int list → int`

która dla danej liczby c i listy l obliczy:

$$\min \{|c + x_i + x_j| : 1 \leq i, j \leq n\}$$

Na przykład, `fit 42 [-28, -25, -15, -1, 4, 8, 15, 60] = 1`, ponieważ $|42 - 28 - 15| = 1$.

18. Zdefiniuj taką procedurę `przedziały: int list → int*int`, że `przedziały [a1; ...; an] = (k, l)`, gdzie (k, l) jest taką parą liczb $1 \leq k \leq l \leq n$, dla której suma $a_k + \dots + a_l$ jest największa.

19. Prostokątną tabelę wypełnioną liczbami całkowitymi reprezentujemy jako listę list liczb całkowitych — każdy wiersz to lista liczb, a tabela to lista wierszy. Oczywiście wszystkie listy reprezentujące wiersze są równej długości.

Rozpatrujemy tabele, których wiersze są uporządkowane ściśle rosnąco, a kolumny ściśle malejąco. Napisz procedurę `znajdz : int list list → int → (int * int) list`, która znajduje wszystkie pozycje, na jakich w takiej tabeli znajduje się podana liczba.

20. Rozważmy następującą metodę kompresji ciągów liczb całkowitych: Jeżeli w oryginalnym ciągu ta sama liczba powtarza się kilka razy z rzędu, to jej kolejne wystąpienia reprezentujemy za pomocą jednej tylko liczby. Konkretnie, i powtórzeń liczby k reprezentujemy w ciągu skompresowanym jako $2^{i-1} \cdot (2 \cdot k - 1)$.

Napisz procedurę `dekompresuj : int list → int list` dekompresującą zadaną listę. Możesz założyć, że lista skompresowana nie zawiera zer.

Podaj specyfikację (warunek początkowy i końcowy) wszystkich zdefiniowanych procedur i uzasadnij **zwięźle** ich pełną poprawność. W przypadku rekurencyjnych procedur iteracyjnych warunek początkowy musi zawierać niezmiennik iteracji.

```
dekompresuj [1; 3; 3; 9; 42; 3];;
```

```
- : int list = [1; 2; 2; 5; 11; 11; 2]
```

21. Palindrom, to taka lista p , że $p = \text{rev } p$. Napisz procedurę `palindrom : α list → int`, która dla danej listy obliczy długość jej najdłuższego spójnego fragmentu, który jest palindromem.
22. Napisz procedurę `podziel : int → 'a list → 'a list list`, która dla $k > 0$ i listy $[x_1; \dots; x_n]$ podzieli ją na listę list postaci:

$$[[x_1; x_{k+1}; x_{2k+1}; \dots]; [x_2; x_{k+2}; x_{2k+2}; \dots]; \dots; [x_k; x_{2k}; x_{3k}; \dots]]$$

Przykład: `podziel 3 [1;2;3;4;5;6;7] = [[1; 4; 7]; [2; 5]; [3; 6]]`.

23. Napisz funkcję `multipart : α list → int list → α list list`, która dla wywołania `multipart l [d1; ...; dn]` dzieli listę l na n części o długościach d_1, d_2, \dots, d_n , tzn. wynikiem powinna być lista list, której pierwszym elementem jest lista składająca się z pierwszych d_1 elementów listy l , drugim elementem jest lista składająca się z kolejnych d_2 elementów listy l , itd. Jeżeli lista l okaże się za krótka należy użyć tyle elementów ile się da. Możesz założyć, że liczby d_1, d_2, \dots, d_n są nieujemne.

Przykłady:

```
multipart [0; 1; 2; 3; 4; 5; 6; 7; 8; 9] [3; 0; 2; 0; 0; 1] = [[0; 1; 2];
[]; [3; 4]; []; []; [5]]
```

```
multipart [0; 1; 2; 3; 4; 5; 6; 7; 8; 9] [3; 0; 2; 0; 0; 1; 8] = [[0; 1; 2];
[]; [3; 4]; []; []; [5]; [6; 7; 8; 9]]
```

24. Dana jest lista nieujemnych liczb całkowitych uporządkowana ściśle rosnąco. Napisz procedurę `kwadraty : int list → int list`, która zwróci uporządkowaną rosnąco listę wszystkich takich liczb x , że zarówno x , jak i x^2 występują na danej liście. Na przykład, `kwadraty [0; 2; 3; 4; 7; 12; 16; 20] = [0; 2; 4]`.

25. [XIII OI] Mały Jaś dostał od rodziców na urodziny nową zabawkę, w której skład wchodzi rurka i krążki. Rurka ma nietypowy kształt — mianowicie jest to połączenie pewnej liczby walców (o takiej samej grubości) z wyciętymi w środku (współosiowo) okrągłymi otworami różnej średnicy. Rurka jest zamknięta od dołu, a otwarta od góry. Krążki w zabawce Jasia są walcami o różnych średnicach i takiej samej grubości co walce tworzące rurkę.

Jaś wymyślił sobie następującą zabawę. Mając do dyspozycji pewien zestaw krążków zastanawia się, na jakiej głębokości zatrzymałby się ostatni z nich, gdyby wrzucać je kolejno do rurki centralnie (czyli dokładnie w jej środek). Każdy kolejny krążek po wrzuceniu spada dopóki się nie zaklinuje (czyli nie oprze się o walec, w którym wycięty jest otwór o mniejszej średnicy niż średnica krążka), albo nie natrafi na przeszkodę w postaci innego krążka lub dna rurki.

Napisz procedurę `krążki : int list → int list → int`, która na podstawie listy średnic otworów w kolejnych walcach tworzących rurkę oraz listy średnic kolejno wrzucanych krążków obliczy głębokość, na której zatrzyma się ostatni wrzucony krążek (lub 0 jeśli nie wszystkie krążki zmieszczą się w rurce).

26. System -2 -kowy jest podobny do systemu binarnego, ale podstawą tego systemu jest liczba -2 . Są dwie możliwe cyfry: 0 i 1. Ciąg cyfr postaci $c_k c_{k-1} \dots c_1 c_0$ reprezentuje liczbę $\sum_{i=0}^k c_i (-2)^i$. Na przykład, liczbę 42 reprezentujemy jako 1111110_{-2} . Kolejne cyfry, od mniej do bardziej znaczących, tworzą listę. Przyjmujemy, że lista pusta reprezentuje 0.

(a) [V OI, zadanie Bankomaty] Napisz procedurę `neg2_of_int : int → int list`, która dla danej liczby całkowitej znajdzie jej przedstawienie w systemie o podstawie -2 .

(b) Napisz procedurę `int_of_neg2 : int list → int`, która dla danej reprezentacji liczby całkowitej w systemie o podstawie -2 znajdzie jej wartość.

(c) Napisz procedurę `inc : int list → int list`, która dla danej listy będącej reprezentacją liczby x , wyznaczy reprezentację liczby $x + 1$.

(d) Napisz procedurę `inv : int list → int list`, która dla danej reprezentacji liczby całkowitej x w systemie -2 -ym obliczy reprezentację liczby $-x$. Na przykład: `przeciwna [1,0,0,1,1] = [1,1,0,1]`.

(e) Napisz procedurę `sum : int list → int list → int list`, która dla danych dwóch reprezentacji liczb całkowitych w systemie -2 -ym obliczy reprezentację ich sumy. Na przykład: `dodawanie [1;0;1;0;0;1;1] [1;0;1] = [0;1;1;1;1;1;1]`.

27. Wielomian postaci $a_0 \cdot x^0 + a_1 \cdot x^1 + \dots + a_n \cdot x^n$ reprezentujemy w postaci listy $[a_0; a_1; \dots; a_n]$. Napisz procedurę `mnoz : float list → float list → float list` mnożącą wielomiany.

Uwaga: Pusta lista reprezentuje zerowy wielomian.

28. Zdefiniuj typ reprezentujący drzewa o wierzchołkach dowolnego (skończonego) stopnia. Zdefiniuj garść procedur operujących na takich drzewach (np. głębokość, liczbę elementów, lista elementów w porządku prefiksowym/postfiksowym).
29. Dana jest deklaracja typu drzew binarnych:

```
type  $\alpha$  tree =
    Node of  $\alpha$  tree *  $\alpha$  *  $\alpha$  tree |
    Null;;
```

Drzewo nazywamy *ultralewicowym* jeśli głębokości kolejnych pustych poddrzew (Null), od lewej do prawej, tworzą ciąg nierosnący. Napisz procedurę `ultraleft : α tree \rightarrow bool`, która sprawdza czy dane drzewo jest ultralewicowe.

30. [II OI, zadanie Obchodzenie drzewa skokami] Jak obejść wszystkie wierzchołki drzewa (ogólnego) tak, aby odległość między kolejnymi wierzchołkami nie przekraczała 3.
31. Rozważmy drzewo binarne, w którego wierzchołkach pamiętane są różne dodatnie liczby całkowite. Dane są dwie listy: wyniki obejścia drzewa w porządku infiksowym i prefiksowym. Napisz procedurę, która odtwarza drzewo na podstawie takich dwóch list.
32. Napisz procedurę, która dla dowolnego drzewa binarnego poprawia je tak, że spełnia ono słabszą wersję warunku BST: dla dowolnego węzła, lewy syn nie jest większy, a prawy nie jest mniejszy, niż węzeł.
33. Napisz funkcję `potnij`, która dla danego predykatu p typu $\alpha \rightarrow \text{bool}$, oraz drzewa binarnego t typu α tree, zwróci następujący wynik. Wybierzmy wszystkie węzły drzewa t zawierające wartości, które spełniają predykat p i usuńmy z drzewa krawędzie łączące te węzły z ich rodzicami (w przypadku korzenia, nie usuwamy żadnej krawędzi). W rezultacie drzewo t może się rozpaść na pewną liczbę mniejszych drzew. `potnij p t` powinno zwrócić listę tych drzew (w dowolnej kolejności).
34. Napisz procedurę, która przekształca dane drzewo binarne w wyważone drzewo binarne, zachowując kolejność elementów w porządku infiksowym.
35. Napisz procedurę `rozcięcie`, która znajduje taką krawędź w danym drzewie binarnym, że po jej usunięciu drzewo rozpada się na dwa drzewa o minimalnej różnicy liczb węzłów. (Wynikiem procedury powinien być dolny koniec szukanej krawędzi.)
36. [PCh] Dana jest deklaracja typu drzew binarnych:

```
type tree = Node of tree * int * tree | Null;;
```

Napisz procedurę `naprzemienne : tree \rightarrow bool`, która sprawdza, czy wartości występujące na każdej ścieżce od korzenia do liści mają naprzemienne znaki. Przyjmujemy

przy tym, że zero jest równocześnie dodatnie i ujemne, tzn. po zerze może następować dowolna liczba. Na przykład, następujące liczby mają naprzemienne znaki: 2, -1, 3, 0, -2, 0, 0, 7, 0, 4.

37. [PCh] Dana jest deklaracja typu drzew binarnych:

```
type tree = Node of tree * int * tree | Null;;
```

Napisz procedurę `symetryczne : tree → bool`, która sprawdza, czy drzewo jest symetryczne (ze względu na odbicie lewo-prawo).

38. Dana jest deklaracja typu reprezentującego wyrażenia logiczne:

```
type expr =  
  And of expr * expr |  
  Or of expr * expr |  
  Not of expr |  
  Value of bool;;
```

Warianty `And`, `Or` i `Not` reprezentują odpowiednie operacje logiczne, natomiast wariant `Value` reprezentuje wartość logiczną `true` lub `false`.

Napisz funkcję `wartościowania : expr → int`, która dla danego wyrażenia policzy na ile różnych sposobów można przypisać wszystkim wariantom `Value` wartości logiczne, tak aby wartość całego wyrażenia wyliczała się do `true`.

Przykład: `wartościowania Or(Value true, Value false) = 3`, gdyż wyrażenie wyliczy się do `true`, w trzech przypadkach: `Or(Value true, Value true)`, `Or(Value true, Value false)`, `Or(Value false, Value true)`.

39. Dana jest deklaracja typu drzew binarnych:

```
type 'a tree = Node of 'a * 'a tree * 'a tree | Null;;
```

Skosokość drzewa jest zdefiniowana w następujący sposób:

- skosokość `Null` wynosi 0,
- skosokość `Node(x, t1, t2)` jest równa $\max(2 \cdot s_1, 2 \cdot s_2 + 1)$, gdzie s_1 i s_2 to skosokości, odpowiednio, t_1 i t_2 .

Dla danego drzewa wolno nam w dowolnych jego węzłach zamieniać miejscami lewe i prawe poddrzewa. Napisz procedurę `skosokosc : α tree → α tree`, która przekształca dane drzewo tak, aby zminimalizować jego skosokość. Na przykład, dla:

```
t = Node(5,  
  Node(4, Null, Node(2, Null, Null)),  
  Node(6,  
    Node(1, Null, Null),  
    Node(3, Null, Null)
```

```

    )
  )

skosokosc t = Node(5,
    Node(6,
        Node(1, Null, Null),
        Node(3, Null, Null)
    ),
    Node(4, Node(2, Null, Null), Null)
)

```

Skosokość tak uzyskanego drzewa wynosi 6.

40. Dana jest deklaracja typu drzew binarnych:

```
type  $\alpha$  tree = Node of  $\alpha$  tree *  $\alpha$  *  $\alpha$  tree | Nil;;
```

Napisz procedurę `liscie` : α tree \rightarrow α list, która tworzy listę wartości znajdujących się w liściach drzewa.

41. Dana jest deklaracja typu drzew binarnych:

```
type tree = Node of tree * int * tree | Null;;
```

Napisz procedurę `czapeczka` : tree \rightarrow tree \rightarrow int, która obliczy na ilu poziomach, idąc od góry, dane drzewa są identyczne, tzn. na tych poziomach drzewa mają ten sam kształt, a w odpowiadających sobie wierzchołkach są takie same liczby.

42. Dany jest typ reprezentujący drzewa binarne:

```
type  $\alpha$  tree =
    Node of  $\alpha$  tree *  $\alpha$  *  $\alpha$  tree |
    Null;;
```

Zdefiniuj typ α przechadzka oraz procedury:

```
buduj:  $\alpha$  tree  $\rightarrow$   $\alpha$  przechadzka
```

```
w_lewo:  $\alpha$  przechadzka  $\rightarrow$   $\alpha$  przechadzka
```

```
w_prawo:  $\alpha$  przechadzka  $\rightarrow$   $\alpha$  przechadzka
```

w_gore: α przechadzka \rightarrow α przechadzka

obejrzyj: α przechadzka \rightarrow α

umożliwiający „przechadzanie” się po drzewie i oglądanie go. Wywołanie `buduj d` powinno konstruować przechadzkę, w której znajdujemy się w korzeniu drzewa `d`. Wywołanie `w_lewo p` powinno konstruować przechadzkę powstałą przez przejście do lewego poddrzewa. Analogicznie powinny działać procedury `w_prawo` i `w_gore`. Procedura `obejrzyj` powinna zwrócić element przechowywany w wierzchołku drzewa, w którym aktualnie jesteśmy.

Podczas przechadzki po drzewie możemy przebywać jedynie w wierzchołkach `Node`. Jeśli nastąpi próba wejścia do pustego drzewa `Null` lub wyjścia „ponad” korzeń, należy podnieść wyjątek `Poza_drzewem`.

43. Dana jest definicja typu: `type tree = Node of tree * tree | Null`. Odległością między dwoma wierzchołkami (`Node`) w drzewie nazywamy minimalną liczbę krawędzi jakie trzeba przejść z jednego wierzchołka do drugiego. Średnicą drzewa nazwiemy maksymalną odległość między dwoma węzłami (`Node`) w drzewie. Przyjmujemy, że średnica pustego drzewa (`Null`) jest równa 0.

Napisz procedurę `średnica : tree -> int`, która oblicza średnicę danego drzewa.

44. [PCh] Dana jest definicja typu drzew binarnych:

```
type  $\alpha$  tree =
```

```
  Node of  $\alpha$  tree *  $\alpha$  *  $\alpha$  tree |
```

```
  Null
```

Powiemy, że węzeł drzewa jest *środkowy*, jeśli ma tyle samo węzłów przodków, co węzłów potomków. Napisz procedurę `srodkowe : α tree \rightarrow α list`, która dla danego drzewa zwróci listę wartości znajdujących się w węzłach środkowych. Wartości na liście wynikowej mogą być w dowolnej kolejności.

45. Dana jest deklaracja typu drzew binarnych, w których węzłach znajdują się liczby całkowite:

```
type tree = Null | Node of int * tree * tree
```

Przekrojem drzewa nazwiemy dowolny taki zbiór jego wierzchołków (`Node`), że na każdej ścieżce od korzenia do liścia (`Null`) znajduje się dokładnie jeden wierzchołek należący do przekroju.

Napisz procedurę `cut : tree \rightarrow int`, która dla danego drzewa wyznaczy najmniejszą możliwą sumę liczb znajdujących się w wierzchołkach tworzących przekrój danego drzewa.

46. Dana jest definicja typu drzew dowolnego stopnia:

```
type  $\alpha$  tree = Node of  $\alpha$  *  $\alpha$  tree list;;
```

Napisz procedurę `liscie` : α tree \rightarrow int list, która dla danego drzewa liczy ile w tym drzewie jest liści na poszczególnych głębokościach i zwraca listę liczb postaci $[g_0; g_1; \dots; g_h]$, gdzie g_i to liczba liści znajdujących się na głębokości i , a h to wysokość drzewa.

47. Zdefiniuj typ reprezentujący dni tygodnia, miesiące i datę. Zdefiniuj procedurę obliczającą na podstawie daty dzień tygodnia. Możesz założyć, że dana jest procedura `sylwester`, która na podstawie roku określa jakiego dnia tygodnia był Sylwester.

Wytyczne dla prowadzących ćwiczenia

To duży zestaw zadań. Należy na nie poświęcić ok. 2 zajęć, a i tak nie wszystkie zdąży się zrobić. Narazie nie korzystamy z procedur wyższych rzędów, nawet gdyby na wykładzie była już o nich mowa. Pozostałe zadania studenci mogą rozwiązać sami powtarzając materiał przed kolokwium.

Ad. 10 Oczekujemy rozwiązania wykonującego co najwyżej $\frac{3n}{2} - 1$ porównań.

Ad. 11 Szukamy rozwiązania o liniowej złożoności czasowej. Należy zwrócić uwagę na uzasadnienie poprawności rozwiązania. Pomaga w tym sformułowanie niezmiennika.

Uwaga: To zadanie można też rozwiązać probabilistycznie, metodą Las Vegas: wylosuj element i sprawdź czy to większość. Oczekiwana złożoność czasowa jest liniowa.

Ad. 21 Oczekujemy rozwiązania o złożoności czasowej $\Theta(n^2)$.

Ad. 40 Oczekujemy rozwiązania o złożoności czasowej $\Theta(n)$, bez względu na kształt drzewa. Należy więc unikać operacji @.

Wykład 5. Moduły

5.1 Wprowadzenie

Każdy duży system powinien być podzielony na mniejsze, łatwiejsze do ogarnięcia składowe. Jednocześnie, interakcje między tymi składowymi powinny być ograniczone do niezbędnego minimum. Stosując taką strukturalizację jesteśmy w stanie ogarnąć, jeśli nie cały system, to jedną składową na raz, wraz ze wszystkimi elementami mającymi wpływ na nią. W przypadku programów, takie składowe to *moduły*.

Każdy moduł ma ściśle określony *interfejs* — zestaw pojęć programistycznych, które realizuje. Jedne moduły mogą korzystać z pojęć zaimplementowanych przez inne. Moduł można określić jako fragment systemu polegający na wykonaniu wyodrębnionego zadania programistycznego. Zwykle sformułowaniu zadania programistycznego towarzyszy (mniej lub bardziej formalna) specyfikacja własności, które powinny posiadać pojęcia programistyczne implementowane przez moduł.

Takie zasady konstrukcji modułów pozwalają (po podzieleniu systemu na moduły) na niezależne ich opracowywanie. Podobnie, moduły można też niezależnie kompilować. Na poziomie języka programowania, wszystkie informacje konieczne do skompilowania modułu są zawarte w interfejsach wykorzystywanych przez niego modułów.

W Ocamlu możemy osobno definiować interfejsy i implementacje modułów. Interfejsy modułów nazywamy *sygnaturami* natomiast ich implementacje *strukturami*.

Dodatkowo istnieje pojęcie modułów sparametryzowanych — *funktorów*. Są to moduły, które mają określony interfejs, a także korzystają z modułów o określonych interfejsach, ale nie sprecyzowanej implementacji. Dopiero po podaniu implementacji uzyskujemy wynikowy moduł. Funktory można traktować jak konstrukcje przekształcające moduły w moduły. Zajmiemy się nimi w wykładzie 12.

5.2 Proste struktury

Definiując strukturę zbieramy razem definicje pojęć, które ta struktura ma udostępniać, otaczamy je słowami `struct ... end` i nadajemy modułowi nazwę.

$$\begin{aligned} \langle \text{definicja} \rangle &::= \text{module } \langle \text{Identyfikator} \rangle \equiv \langle \text{struktura} \rangle \\ \langle \text{struktura} \rangle &::= \text{struct } \{ \langle \text{definicja} \rangle \}^* \text{end} \end{aligned}$$

Tak zdefiniowana struktura udostępnia wszystkie pojęcia zdefiniowane wewnątrz — ma najszerszy z możliwych interfejsów. Do pojęć zdefiniowanych wewnątrz struktury możemy dostać się stosując nazwy kwalifikowane postaci:

$$\langle \text{identyfikator} \rangle ::= \langle \text{Identyfikator} \rangle _ \langle \text{identyfikator} \rangle$$

Możemy też „otworzyć” strukturę, tzn. wyłuskać z niej wszystkie udostępniane przez nią pojęcia tak, aby były dostępne bez kwalifikowania nazwą modułu.

$$\langle \text{jednostka kompilacji} \rangle ::= \text{open } \langle \text{Identyfikator} \rangle$$

Przykład: Proste przykłady modułów:

```
module Modulik =
  struct
    type typik = int list
    let lista = [2; 3; 7]
    let prod l = fold_left ( * ) 1 l
  end;;

Modulik.prod Modulik.lista;;
open Modulik;;
prod lista;;

module Pusty =
  struct
  end;;
```

Przykład: Moduły mogą zawierać wewnątrz moduły:

```
module M =
  struct
    module A =
      struct
        let a = 27
      end
    module B =
      struct
        let b = 15
      end
  end;;

M.A.a + M.B.b;;
```

Struktury można rozszerzać. Definiując jedną strukturę można „wciągnąć” do niej zawartość innej struktury.

$\langle \text{definicja} \rangle ::= \text{include } \langle \text{Identyfikator} \rangle \mid \dots$

Przykład:

```
module Mod =
  struct
    include Modulik
    let n = List.length lista
  end;;
```

5.3 Sygnatury

Sygnatura, to interfejs modułu — określa, które elementy struktury mają być dostępne na zewnątrz. Wszystko czego nie widać w sygnaturze jest ukryte. Sygnatura może zawierać deklaracje:

- wartości, z podaniem typu wartości,
- typu wraz z jego definicją,
- typu abstrakcyjnego (bez podania definicji),
- sygnatury lokalnej
- wyjątków.

```
⟨definicja⟩ ::= module type ⟨Identyfikator⟩ ≡ ⟨sygnatura⟩
⟨sygnatura⟩ ::= sig {⟨deklaracja⟩}* end
⟨deklaracja⟩ ::= type {⟨parametr typowy⟩}* ⟨identyfikator⟩ [≡ ⟨typ⟩] |
                 val ⟨identyfikator⟩ : ⟨typ⟩ |
                 module type ⟨Identyfikator⟩ ≡ ⟨sygnatura⟩ |
                 exception ⟨wariant⟩ |
                 ...
```

Przykład: Taka sobie sygnatura:

```
module type S =
sig
  type abstrakcyjny
  type konkretny = int * float
  val x : abstrakcyjny * konkretny
  module type Pusty = sig end
  exception Wyjatek of abstrakcyjny
end;;
```

Przykład: Sygnatura kolejek FIFO:

```
module type FIFO =
sig
  exception EmptyQueue
  type 'a queue
  val empty : 'a queue
  val insert : 'a queue ->'a ->'a queue
  val front : 'a queue ->'a
  val remove : 'a queue ->'a queue
end;;
```

Sygnatury można rozszerzać. Definiując jedną sygnaturę można „wciągnąć” do niej zawartość innej sygnatury.

$$\langle \text{deklaracja} \rangle ::= \text{include } \langle \text{Identyfikator} \rangle \mid \dots$$

Przykład: Sygnatura kolejek dwustronnych:

```
module type QUEUE =
  sig
    include FIFO
    val back : 'a queue ->'a
    val insert_front : 'a queue ->'a ->'a queue
    val remove_back : 'a queue ->'a queue
  end;;
```

Sygnatury można ukonkretniać. Na przykład, jeżeli sygnatura zawiera typ abstrakcyjny, to można go ukonkretnić. Można podać jaka ma być jego implementacja.

$$\langle \text{sygnatura} \rangle ::= \langle \text{sygnatura} \rangle \text{ with type } \{ \langle \text{parametr typowy} \rangle \}^* \langle \text{identyfikator} \rangle \equiv \langle \text{typ} \rangle \mid \dots$$

Przykład:

```
module type LIST = FIFO with type 'a queue = 'a list;;
```

5.4 Łączenie struktur i sygnatur

Podając jawnie sygnaturę dla struktury ograniczamy wgląd do środka struktury. Można to zrobić na kilka sposobów: treść sygnatury i struktury można podać *explicite* lub odwołać się do zdefiniowanej sygnatury/struktury.

$$\begin{aligned} \langle \text{definicja} \rangle &::= \text{module } \langle \text{Identyfikator} \rangle [_ : \langle \text{sygnatura} \rangle] \equiv \langle \text{struktura} \rangle \\ \langle \text{struktura} \rangle &::= \langle \text{struktura} \rangle : \langle \text{sygnatura} \rangle \\ \langle \text{struktura} \rangle &::= \langle \text{Identyfikator} \rangle \\ \langle \text{sygnatura} \rangle &::= \langle \text{Identyfikator} \rangle \end{aligned}$$

Przykład: Różne sposoby definiowania modułu FIFO:

```
module Fifo_implementation =
  struct
    exception EmptyQueue
    type 'a queue = 'a list
    let empty = []
    let insert q x = q @ [x]
    let front q =
      match q with
      [] ->raise EmptyQueue |
      h::_ ->h
    let remove q =
      match q with
      [] ->raise EmptyQueue |
      _::t ->t
  end;;

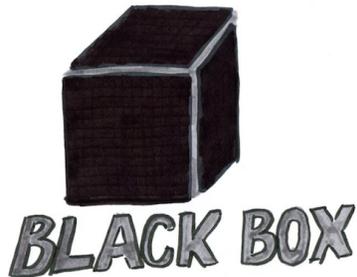
module Fifo : FIFO = Fifo_implementation;;
module Fifo = (Fifo_implementation : FIFO);;

module Fifo : sig ... end = struct ... end;;
```

5.5 Jak wyodrębnić moduły?

Jakimi zasadami kierować się dzieląc program na moduły? Każdy program można podzielić na moduły: pierwsze 100 linii, drugie 100 linii, itd. Oczywiście nie każdy podział jest właściwy. Dzieląc program na moduły powinniśmy kierować się następującymi zasadami:

- Moduł ma charakter czarnej skrzynki (ang. *black-box approach*). Na zewnątrz modułu widoczne są wyłącznie te pojęcia programistyczne, które tworzą interfejs.
- Natomiast sposób ich implementacji, jak i ew. pojęcia pomocnicze są ukryte wewnątrz modułu (ang. *information hiding*).
- Specyfikacja modułu nie powinna odwoływać się do sposobu implementacji modułu, a jedynie do takich właściwości modułu, które może zaobserwować użytkownik modułu. Co więcej, użytkownik nie tylko nie ma możliwości, ale i nie powinien (w swym programie) wnikać w sposób implementacji modułu (ang. *separation of concerns*).
- Powiązania między modułami były jak najmniejsze, aby jak najmniej szczegółów budowy jednego modułu miało wpływ na budowę innego modułu.
- Jeden moduł powinien koncentrować się na jednej decyzji projektowej, tzw. „sekrete” modułu. Nie należy łączyć nie związanych ze sobą sekretów w jednym module. Dobrym kandydatem na sekret modułu jest sposób realizacji struktury danych (wraz z operacjami na niej).

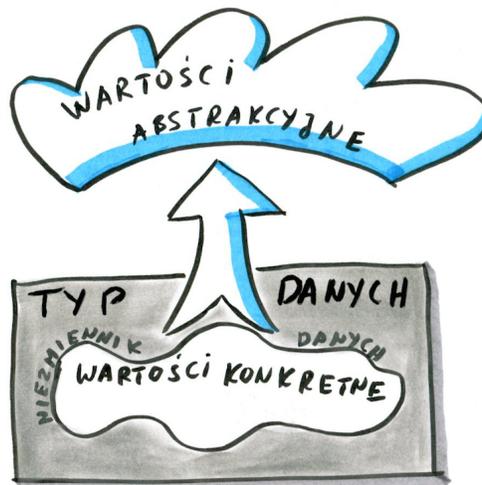


Rozważając kilka możliwych podziałów na moduły możemy porównać je stosując następujący eksperyment myślowy. Przygotowujemy listę potencjalnych zmian w programie. Lista ta nie może być wydumana, ani nie może to być lista zmian, które łatwo wprowadzić do programu, ale lista realnych zmian, które mogą wynikać z potrzeb użytkownika programu. Dla każdej z tych zmian i każdego z podziałów na moduły badamy ile modułów należy zmodyfikować w celu wprowadzenia danej zmiany. Im więcej modułów, tym gorzej.

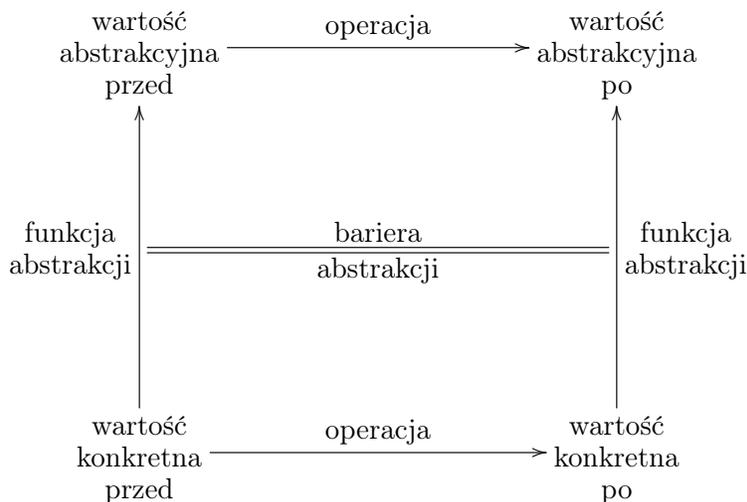
5.6 Moduły zorientowane wokół danych

Zastanówmy się, jak tworzyć moduły, których sekretem jest sposób realizacji pewnej struktury danych. Musimy podjąć kilka decyzji projektowych:

- jaki interfejs powinien być udostępniony użytkownikowi,
- jaki jest zbiór wartości, które chcemy reprezentować, tzw. wartości abstrakcyjnych,
- jaka struktura danych będzie odpowiednia do reprezentowania wartości abstrakcyjnych, tzw. wartości konkretne, (tj. jaki będzie typ danych i które z wartości tego typu będą poprawnymi reprezentacjami wartości abstrakcyjnych, tzw. niezmiennik struktury danych),
- mając daną wartość konkretną, jaka jest odpowiadająca jej wartość abstrakcyjna, tzw. funkcja abstrakcji,



Wszystkie one określają sposób implementacji struktury danych. Odpowiednio dobrany interfejs tworzy pewną *abstrakcję danych* w której jej funkcjonalność jest oddzielona od implementacji. Powyżej tej bariery posługujemy się *wartościami abstrakcyjnymi*, a poniżej *wartościami konkretnymi*. Wartości abstrakcyjne, to takie, jakie widzi użytkownik typu, a konkretne to sposób ich implementacji.

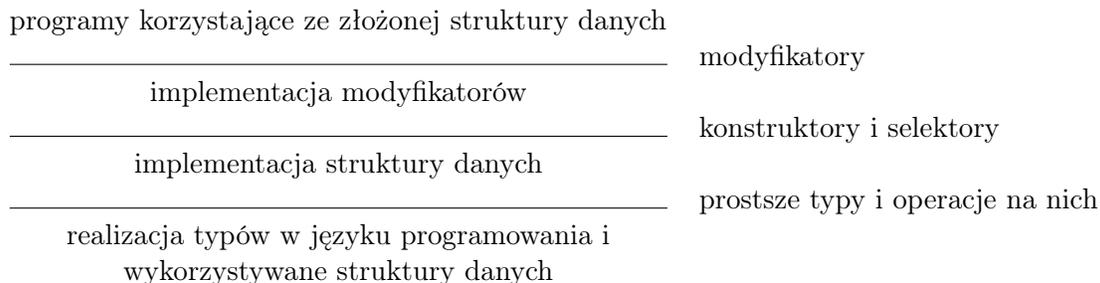


Operacje tworzące interfejs możemy podzielić na trzy kategorie:

- konstruktory — tworzą złożone wartości z prostszych,
- selektory — wyłuskują elementy lub badają cechy złożonych wartości,
- modyfikatory — przekształcają złożone wartości.

Odpowiednio dobrane konstruktory i selektory mogą tworzyć dodatkową abstrakcję, za pomocą której są zaimplementowane modyfikatory. Wówczas implementacja modyfikatorów korzysta wyłącznie z wartości abstrakcyjnych i jest niezależna od implementacji struktury danych. Zawsze, im mniejsze fragmenty kodu są wrażliwe na potencjalne zmiany tym lepiej.

W ten sposób ustalone interfejsy i abstrakcje tworzą bariery oddzielające różne poziomy abstrakcji.



5.7 Przykład: Pakiet liczb wymiernych

5.7.1 Abstrakcja danych

Spróbujmy zaimplementować zgodnie z opisanym podejściem pakiet arytmetyki liczb wymiernych. Do zaimplementowania samej struktury danych potrzebujemy jednego konstruktora i

dwóch selektorów:

```
module type ULAMKI =
  sig
    type t (* typ abstrakcyjny reprezentujący ułamki *)

    val ulamek : int ->int ->t
    (* konstruktor,ulamek l m tworzy ułamek  $\frac{l}{m}$  przy założeniu, że  $m \neq 0$  *)

    val licznik : t ->int
    (* selektor,zwraca licznik ułamka *)

    val mianownik : t ->int
    (* selektor,zwraca mianownik ułamka *)
  end;;
```

Wymagamy przy tym, aby zachodziło:

$$\forall_{l,m \in \mathbb{N}, m \neq 0} \frac{\text{licznik}(\text{ulamek } l \ m)}{\text{mianownik}(\text{ulamek } l \ m)} = \frac{l}{m}$$

Dowolna implementacja spełniająca te warunki będzie nas satysfakcjonować. Użytkownik pakietu może polegać na *specyfikacji*, ale nie może nic zakładać o jej *implementacji*. Stosujemy zasadę *pobożnych życzeń* i odkładamy implementację tych trzech operacji na później. Zakładamy tylko, że znajduje się ona w module o nazwie `Ulamki`.

```
module Ulamki : ULAMKI = struct
  :
end
```

5.7.2 Operacje arytmetyczne i równość

Oprócz struktury danych musimy zaimplementować operacje arytmetyczne (modyfikatory) i równość.

```
module type RAT =
  sig
    include ULAMKI

    val rowne : t ->t ->bool    (* porównanie *)

    val plus : t ->t ->t      (* suma *)

    val minus : t ->t ->t    (* różnica *)

    val razy : t ->t ->t     (* iloczyn *)

    val podziel : t ->t ->t  (* iloraz *)
  end;;
```

Możemy je zaimplementować zgodnie z następującymi tożsamościami:

$$\frac{l_1}{m_1} + \frac{l_2}{m_2} = \frac{l_1 m_2 + l_2 m_1}{m_1 m_2}$$

$$\frac{l_1}{m_1} - \frac{l_2}{m_2} = \frac{l_1 m_2 - l_2 m_1}{m_1 m_2}$$

$$\frac{l_1}{m_1} \cdot \frac{l_2}{m_2} = \frac{l_1 l_2}{m_1 m_2}$$

$$\frac{l_1/m_1}{l_2/m_2} = \frac{l_1 m_2}{l_2 m_1} \quad \text{dla } l_2 \neq 0$$

$$\frac{l_1}{m_1} = \frac{l_2}{m_2} \Leftrightarrow l_1 \cdot m_2 = l_2 \cdot m_1 \quad \text{dla } m_1 \neq 0 \text{ i } m_2 \neq 0$$

```
module Rat : RAT = struct
  include Ułamki

  let plus x y =
    ulamek
      (licznik x * mianownik y + licznik y * mianownik x)
      (mianownik x * mianownik y);;

  let minus x y =
    ulamek
      (licznik x * mianownik y - licznik y * mianownik x)
      (mianownik x * mianownik y);;

  let razy x y =
    ulamek
      (licznik x * licznik y)
      (mianownik x * mianownik y)

  let podziel x y =
    ulamek
      (licznik x * mianownik y)
      (mianownik x * licznik y)

  let rowne x y =
    licznik x * mianownik y = licznik y * mianownik x

end;;
```

Dzięki zastosowaniu podwójnej bariery abstrakcji, implementacja modyfikatorów i równości nie zależy od sposobu implementacji struktury danych.

5.7.3 Reprezentacja liczb wymiernych

Chcemy teraz określić **reprezentację** liczb wymiernych. W tym celu musimy określić:

- określić **niezmiennik struktury danych**, czyli zbiór wartości **konkretnych** reprezentujących liczby wymierne,
- określić **funkcję abstrakcji**, która każdej konkretnej wartości przyporządkowuje reprezentowaną przez nią wartość **abstrakcyjną**,
- wyspecyfikować i zaimplementować procedury określające reprezentację struktury danych (**ulamek**, **licznik**, **mianownik**).

Najprostsza reprezentacja polega na pamiętaniu pary: (licznik, mianownik). Wartości konkretne to wszystkie takie pary (l, m) , w których $m \neq 0$. Funkcja abstrakcji parze (l, m) przyporządkowuje ułamek $\frac{l}{m}$. Operacje **ulamek**, **mianownik** i **licznik** muszą spełniać specyfikacje analogiczne do tych wyrażonych za pomocą wartości abstrakcyjnych. Można je więc zdefiniować tak:

```
module Ulamki : ULAMKI = struct
  type t = int * int
  let ulamek l m = (l,m)
  let licznik (l,_) = l
  let mianownik (_,m) = m
end;;
```

5.7.4 Zmiana reprezentacji liczb wymiernych

Przypuśćmy, że chcemy zmienić reprezentację na taką, w której pamiętamy parę (licznik, mianownik), ale ułamka w postaci skróconej. Tak więc, nasz zbiór wartości konkretnych obejmuje takie pary (l, m) , w których $m \neq 0$ oraz l i m są względnie pierwsze. Funkcja abstrakcji pozostaje bez zmian. Konstruktor **ulamek** musi dodatkowo skracać ułamki. Operacje **ulamek**, **mianownik** i **licznik** można zdefiniować tak:

```
module Ulamki : ULAMKI = struct
  type t = int * int
  let ulamek l m =
    let r = nwd l m
    in ((l / r), (m / r))
  let licznik (l,_) = l
  let mianownik (_,m) = m
end;;
```

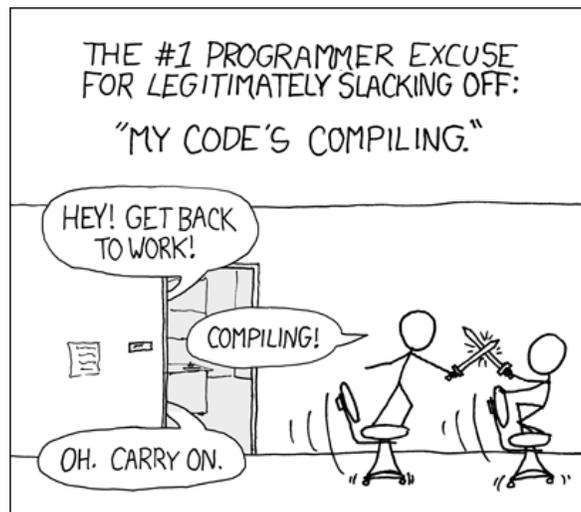
Tę reprezentację można dalej ulepszać biorąc pod uwagę znaki licznika i mianownika.

5.8 Podsumowanie wprowadzonych zasad

- **Najpierw celuj, potem strzelaj** — najpierw określamy specyfikację, a potem do niej dorabiamy implementację, a nie odwrotnie,

- **Technika pobożnych życzeń** — rozwiązując problem top-down dzielimy go na pod-problemy, zakładamy chwilowo, że podproblemy są rozwiązane i rozwiązujemy główny problem, potem przystępujemy do rozwiązania podproblemów,
- **Dekompozycja i abstrakcja** — problem powinniśmy dzielić na podproblemy zgodnie z jego naturą, wyodrębniając jak najbardziej uniwersalne podproblemy,
- **Zasady tworzenia modułów** — omówione w niniejszym wykładzie.

5.9 Deser



<http://xkcd.com/303/>

Ćwiczenia

1. [D.L.Parnas, *On the Criteria To Be Used in Decomposing Systems into Modules*, CACM 12 (15), 1972] Rozważmy następujący problem. Należy napisać program, który:

- wczyta plik tekstowy,
- dany plik składa się z wierszy, a każdy wiersz ze słów oddzielonych białymi znakami,
- rozważamy rotacje cykliczne wierszy, powstałe przez przestawienie pewnej liczby słów z początku wiersza na koniec,
- program ma wypisać wszystkie możliwe rotacje cykliczne wierszy z wejścia, bez powtórzeń, w porządku leksykograficznym, oddzielając słowa w wierszach pojedynczymi odstępami.

Podaj jak byś podzielił program na moduły. Wymyśl zestaw realistycznych modyfikacji, jakie należy wprowadzić (a jeszcze lepiej poproś o to kolegę) i sprawdź, jak sprawuje się wymyślony przez Ciebie podział na moduły.

2. Zdefiniuj sygnatury/struktury odpowiadające podanym poniżej pojęciom. Staraj się wykorzystać zdefiniowane wcześniej pojęcia.

- (a) Sygnaturę półgrupy (typ elementów i operacja łączna).
- (b) Sygnaturę monoidu (półgrupa z elementem neutralnym). Monoid:
 - liczb całkowitych z dodawaniem,
 - funkcji (na liczbach całkowitych) z operacją składania,
 - list z operacją sklejaną,
 - macierzy 2×2 z operacją mnożenia.
- (c) Sygnatura grupy (monoid z elementami odwrotnymi).
- (d) Sygnatura pierścienia (dodawanie, mnożenie, jedynka, zero, elementy przeciwne).
- (e) Pierścień liczb całkowitych.
- (f) Sygnatura ciała (pierścień z elementami odwrotnymi).
- (g) Ciało liczb rzeczywistych.
- (h) Sygnatura relacji binarnej na elementach tego samego zbioru (np. relacja równoważności, porządek). Przykładowe relacje.
- (i) Sygnatura porządku z operacjami kresu górnego i dolnego.
- (j) Porządek liniowy na liczbach całkowitych z operacjami \max i \min .

Wytyczne dla prowadzących ćwiczenia

Ad. 1 W zadaniu o podziale programu na moduły należy tak pokierować dyskusją, aby:

- uzyskać przynajmniej dwa istotnie różne podziały na moduły, jeden zorganizowany wokół kolejnych faz przetwarzania lub algorytmów, a drugi wokół struktur danych,
- wśród potencjalnych zmian uwzględnić:
 - nie będziemy odróżniać wielkich i małych liter,
 - zmianę pojęcia znaków (np. przejście na unicode) przy zachowaniu specyficznego porządku leksykograficznego,
 - zmianę sposobu podziału wiersza na słowa lub pliku na wiersze (kodowanie znaków końca wiersza),
 - zamiast rotacji cyklicznych będziemy chcieli przetwarzać inne pochodne ciągi znaków, np. prefiksy lub sufiksy,
 - zamiast standardowego wejścia/wyjścia, program ma przetwarzać dane z/do pliku.

Wykład 6. Procedury wyższych rzędów jako abstrakcje konstrukcji programistycznych

W tym wykładzie zajmiemy się "procedurami wyższych rzędów", tzn. procedurami przetwarzającymi procedury. Przedstawimy na przykładach, w jaki sposób można używać procedur, których argumentami i/lub wynikiem są procedury jako abstrakcja powtarzających się schematów postępowania.

Wcześniej zaznaczyliśmy, że procedury są w funkcyjnych językach programowania obywatelami pierwszej kategorii, czyli są równie dobrymi wartościami jak wszystkie inne. W szczególności procedury mogą być argumentami procedur lub ich wynikami. Argumenty proceduralne występują również w imperatywnych językach programowania (np. w standardzie Pascala). Natomiast to, że procedury mogą być wynikami procedur, jest czymś charakterystycznym dla języków funkcyjnych².

6.1 Typy proceduralne i polimorfizm

Przyjrzyjmy się kilku prostym przykładom procedur wyższych rzędów.

```
let p f = function x ->f x + f (2 * x);;
val p : (int ->int) ->int ->int = <fun>
```

Argumentem procedury `p` jest procedura `f`, natomiast jej wynikiem jest procedura będąca wartością λ -abstrakcji. Zwróćmy uwagę na typ procedury `p`. Możemy go przeczytać jako `(int -> int) -> (int -> int)`. Zarówno argument `f`, jak i wynik `p` są procedurami typu `int -> int`.

```
let twice f = function x ->f (f x);;
val twice : ('a ->'a) ->'a ->'a = <fun>
```

```
twice (function x ->x * (x+1)) 2;;
- : int = 42
```

```
twice (function s ->"mocium panie," ^ s) "me wezwanie";;
- : string = "mocium panie,mocium panie,me wezwanie"
```

Argumentem procedury `twice` jest procedura `f`, natomiast jej wynikiem jest złożenie `f` z samą sobą. Zwróćmy uwagę na typ procedury `twice`. Typ tej procedury czytamy jako: `('a -> 'a) -> ('a -> 'a)`. Kompilator jest w stanie wywnioskować, że `f` jest procedurą i że typ jej argumentu musi być taki sam, jak typ jej wyniku (inaczej nie można by jej złożyć samej ze sobą). Natomiast nie wie jaki to jest typ. Tak naprawdę, może to być dowolny typ.

Mamy tu do czynienia z *polimorfizmem* — ta sama procedura może być **wielu typów**. Oznaczenie `'a` jest tzw. *zmienną typową*. Jeśli w typie występują zmienne typowe, to taki typ jest schematem opisującym wiele typów. Do takiego typu-schematu pasuje każdy taki typ, który możemy z niego uzyskać, podstawiając (równocześnie) za zmienne typowe dowolne typy. Przy tym za różne zmienne typowe możemy podstawiać różne typy, ale za wszystkie

²To, że w językach imperatywnych procedury mogą być przekazywane jako argumenty, a nie mogą być wynikami, wynika ze sposobu kompilacji programów imperatywnych.

wystąpienia tej samej zmiennej typowej musimy podstawiać te same typy. Na przykład, podstawiając za 'a typ int, uzyskujemy z typu ('a -> 'a) -> ('a -> 'a) typ (int -> int) -> (int -> int). Natomiast podstawiając za 'a typ 'a -> 'a, uzyskujemy typ (('a -> 'a) -> ('a -> 'a)) -> (('a -> 'a) -> ('a -> 'a)).

Uwaga: Mechanizm podstawiania typów za zmienne typowe jest dokładnie taki sam, jak podstawianie termów, za zmienne w termach.

Przykład: Spójrzmy na następujące zastosowanie `twice`:

```
let czterokrotnie f = twice twice f;;
val czterokrotnie : ('a ->'a) ->'a ->'a = <fun>
```

Procedura `czterokrotnie` działa w następujący sposób:

$$\text{czterokrotnie } f = (\text{twice twice}) f = \text{twice } (\text{twice } f) = \text{twice } f^2 = f^4$$

Zwróćmy uwagę, że dwa wystąpienia `twice` w definicji `czterokrotnie` mają różne typy. Drugie wystąpienie operuje na procedurze `f`, a więc jest typu ('a -> 'a) -> 'a -> 'a. Natomiast pierwsze wystąpienie `twice` przetwarza drugie, a więc jest typu (('a -> 'a) -> 'a -> 'a) -> ('a -> 'a) -> 'a -> 'a. Jest to możliwe dzięki polimorficzności procedury `twice`.

Składanie procedur możemy zdefiniować następująco:

```
let compose f g = function x ->f (g x);;
val compose : ('a ->'b) ->('c ->'a) ->'c ->'b = <fun>
```

Zwróćmy uwagę, że mamy tu aż trzy zmienne typowe. Mamy tu następujące zależności między typami składowych:

- wynik `g` i argument `f` muszą być tego samego typu,
- argument `g` i argument wyniku są tego samego typu,
- wynik `f` i wynik wyniku `compose` są tego samego typu.

Procedurę `twice` możemy teraz zdefiniować w następujący sposób:

```
let twice f = compose f f;;
val twice : ('a ->'a) ->'a ->'a = <fun>
```

Możemy też zdefiniować wielokrotne składanie funkcji samej ze sobą:

```
let id x = x;;
val id : 'a ->'a = <fun>

let rec iterate n f =
  if n = 0 then id else compose (iterate (n-1) f) f;;
val iterate : int ->('a ->'a) ->'a ->'a = <fun>
```

6.2 Czy istnieją procedury wieloargumentowe?

Typy proceduralne są postaci `argument -> wynik`. Ponieważ zarówno argument, jak i wynik może być typu proceduralnego, więc typ może zawierać więcej „strzałek”. Zastanówmy się nad typem procedur wieloargumentowych. Rozważmy następujące dwie definicje:

```
let plus (x,y) = x + y;;
val plus : int * int ->int = <fun>
```

```
let plus x y = x + y;;
val plus : int ->int ->int = <fun>
```

Pierwsza procedura ma jeden argument, który jest parą liczb całkowitych. Druga procedura ma dwa argumenty będące liczbami całkowitymi. Jej typ można jednak zinterpretować tak: `int -> (int -> int)`. Można ją więc traktować jak procedurę **jednoargumentową**, której wynikiem jest procedura jednoargumentowa, której wynikiem jest suma. Inaczej mówiąc, procedura ta bierze argumenty na raty. Przedstawiona powyżej definicja jest równoważna następującej, lepiej oddającej możliwość brania argumentów po jednym:

```
let plus = function x ->function y ->x + y;;
val plus : int ->int ->int = <fun>
```

Można więc powiedzieć, że istnieją wyłącznie procedury jednoargumentowe, a procedury wieloargumentowe są tak naprawdę procedurami wyższego rzędu. Taki sposób patrzenia na procedury wieloargumentowe może być wygodny. Jeśli kolejność argumentów jest odpowiednio dobrana, to możemy czasem podawać tylko część z nich, otrzymując w wyniku potrzebną procedurę.

```
let plus x y = x + y;;
val plus : int ->int ->int = <fun>
```

```
let inc x = plus 1 x;;
val inc : int ->int = <fun>
```

```
let inc = plus 1;;
val inc : int ->int = <fun>
```

Obie przedstawione formy przekazywania argumentów są sobie równoważne. Dowodem na to są poniższe dwie procedury przekształcające jedną postać w drugą i odwrotnie. Jakiego typu są te procedury?

```
let curry f = function x ->function y ->f (x,y);;
let uncurry f = function (x,y) ->f x y;;
```

Standardową postacią podawania argumentów procedury jest „curry”. Tak więc przedstawione przykłady można zdefiniować i tak:

```

let twice f x = f (f x);;
val twice : ('a ->'a) ->'a ->'a = <fun>

let compose f g x = f (g x);;
val compose : ('a ->'b) ->('c ->'a) ->'c ->'b = <fun>

let curry f x y = f (x,y);;
val curry : ('a * 'b ->'c) ->'a ->'b ->'c = <fun>

let uncurry f (x,y) = f x y;;
val uncurry : ('a ->'b ->'c) ->'a * 'b ->'c = <fun>

```

6.3 Sumy częściowe szeregów

Powiedzmy, że interesuje nas przybliżanie szeregów przez obliczanie ich sum częściowych. Możemy zdefiniować procedurę obliczającą sumę częściową dowolnego szeregu. Jej parametrem jest procedura zwracająca określony element szeregu.

```

let szereg f n =
  let rec sum a n =
    if n = 0 then a else sum (a +. f n) (n - 1)
  in
  sum 0.0 n;;
val szereg : (int ->float) ->int ->float = <fun>

```

Powiedzmy, że chcemy przybliżać szereg $\sum_{i=1}^{\infty} \frac{1}{(4i-3)(4i-1)} = \frac{\pi}{8}$. Żeby móc liczyć sumy częściowe tego szeregu wystarczy, że opiszemy jego elementy.

```

let szereg_pi_8 n =
  szereg
  (function i ->1. /. ((4. *. float i -. 3.) *. (4. *. float i -. 1.)))
  n;;
val szereg_pi_8 : int ->float = <fun>

```

```

let pi = 8. *. szereg_pi_8 1000;;
val pi : float = 3.14109265362103818

```

Procedura `szereg` może służyć do obliczania sum częściowych dowolnych szeregów. Żeby jednak dokonać takiej abstrakcji, musi ona mieć parametr proceduralny.

Przypomnienie hasła „abstrakcja proceduralna” — tutaj abstrahujemy nie procedury pomocnicze, ale wspólny schemat „procedury głównej”, która operuje na różnych procedurach lokalnych.

6.4 Różniczkowanie funkcji

Zdefiniujmy najpierw różniczkę:

```

let rozniczka f x dx = (f (x +. dx) -. f x) /. dx;;
val rozniczka : (float ->float) ->float ->float ->float = <fun>

```

Wówczas pochodną możemy przybliżyć następująco:

```
let pochodna f x = rozniczka f x epsilon;;  
val pochodna : (float ->float) ->float ->float = <fun>
```

Zwróćmy uwagę, że `pochodna` jest procedurą jednoargumentową i wynikiem (`pochodna f`) jest funkcja będąca pochodną `f`. Czyli na procedurę `pochodna` możemy patrzeć albo jak na procedurę dwuargumentową, której wynikiem jest przybliżenie pochodnej danej funkcji w punkcie, albo jak na procedurę jednoargumentową, której wynikiem jest funkcja będąca pochodną danej funkcji.

```
pochodna (function x ->x) 42.69;;  
- : float = 1.000000082740371
```

```
let g = pochodna (function x ->7.0 *. x *. x +. 5.0);;  
val g : float ->float = <fun>
```

```
g 3.0;;  
- : float = 41.9999992118391674
```

6.5 Szukanie zer

Przedstawimy dwie metody szukania zer funkcji:

- przez bisekcję i
- ogólną metodę Newtona (stycznych).

W przypadku metody przez bisekcję mamy dany przedział, na końcach którego funkcja przyjmuje przeciwne znaki. Badając znak funkcji w środku jesteśmy w stanie zawęzić przedział o połowę.

Metoda Newtona, nazywana też metodą stycznych, polega na iterowaniu następującego procesu: w punkcie będącym aktualnym przybliżeniem zera funkcji rysujemy styczną do wykresu funkcji; kolejnym przybliżeniem jest punkt przecięcia stycznej z osią X .

Obie metody polegają na polepszaniu pewnego przybliżenia, aż do uzyskania satysfakcjonującego wyniku.

```
let rec iteruj poczatek popraw czy_dobre wynik =  
  if czy_dobre poczatek then  
    wynik poczatek  
  else  
    iteruj (popraw poczatek) popraw czy_dobre wynik;;  
val iteruj : 'a ->('a ->'a) ->('a ->bool) ->('a ->'b) ->'b = <fun>
```

Jest to ogólny schemat opisanego procesu iteracyjnego. Porównajmy poszczególne składowe tego procesu w przypadku obu metod:

Parametr	Metoda Newtona	Metoda przez bisekcję
poczatek	punkt w pobliżu zera	końce przedziału zawierającego zero
popraw	przecięcie stycznej do wykresu funkcji z osią X	podział przedziału na pół
czy_dobre	wartość funkcji w punkcie jest bliska zeru	wartość funkcji na końcu przedziału jest bliska zeru
wynik	dany punkt	koniec przedziału

6.5.1 Metoda przez bisekcję

Mamy daną funkcję f oraz dwa punkty, l i p , w których funkcja f przyjmuje wartości przeciwnych znaków. Gdzieś pomiędzy l i p funkcja f ma zero. Staramy się przybliżyć ten punkt.

Uproszczenie: przyjmujemy, że $f l \leq 0 < f p$, choć dopuszczamy aby $l < p$ lub $l > p$.

```
let bisekcja f l p =
  let rec szukaj l p = ...
  in
    let wartosc_l = f l
    and wartosc_p = f p
    in
      if ujemne wartosc_l && dodatnie wartosc_p then
        szukaj l p
      else
        szukaj p l;;
val bisekcja : (float ->float) ->float ->float ->float = <fun>
```

Wykorzystujemy procedurę iteruj.

```
let szukaj l p =
  let czy_dobre x = ...
  and popraw x = ...
  in
    iteruj
      (l,p)
      popraw
      czy_dobre
      fst
```

Pozostały do zdefiniowania elementy procesu iteracyjnego. Pomysł polega na zbadaniu znaku funkcji f po środku między l i p . Zależnie od tego znaku, zawężamy przedział poszukiwań o połowę.

```
let czy_dobre x =
  abs_float (f (fst x)) <epsilon

and popraw x =
  let l = fst x
  and p = snd x
  in
    let srodek = average l p
```

```

in
  if dodatnie (f srodek) then
    (1,srodek)
  else
    (srodek,p)

```

Zwróćmy uwagę, że procedura popraw jest procedurą lokalną, znajdującą się wewnątrz procedury szukaj. Dzięki temu ma dostęp do funkcji f .

Oto przykład zastosowania szukania zer przez bisekcję do pierwiastkowania liczb:

```

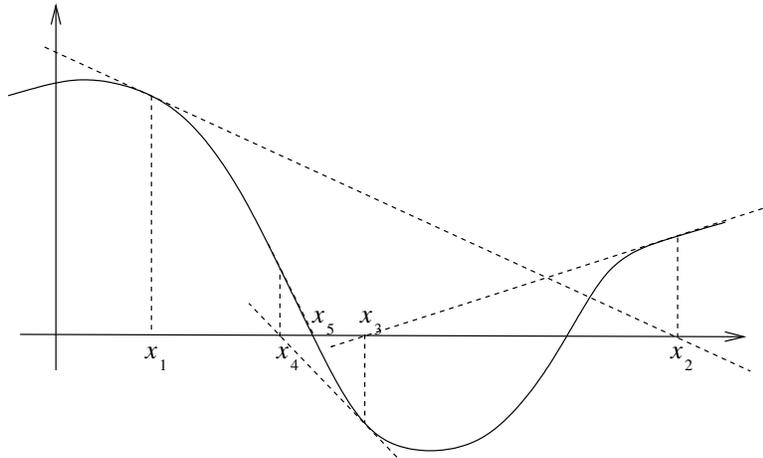
let sqrt a =
  let f x = a -. square x
  in bisekcja f 0.0 (a +. 1.0);;
val sqrt : float ->float = <fun>

sqrt 42.0;;
- : float = 6.48074069840786

```

6.6 Metoda Newtona

Dla uproszczenia, zakładamy, że dana funkcja jest różniczkowalna. Pomijamy kwestię własności stopu przedstawionego algorytmu i ewentualnego dzielenia przez 0. Metoda Newtona działa w następujący sposób: w punkcie będącym aktualnym przybliżeniem zera funkcji rysujemy styczną do wykresu funkcji; kolejnym przybliżeniem jest punkt przecięcia stycznej z osią X .



Zastosujmy teraz procedurę `iteruj` do zaimplementowania metody stycznych Newtona. Zauważmy, że jeżeli naszym przybliżeniem zera jest x , to styczna do wykresu funkcji przecina oś X w punkcie $x - \frac{(f\ x)}{(f'\ x)}$.

```

let id x = x;;
val id : 'a ->'a = <fun>

let newton f x =
  let p = pochodna f

```

```

in
  let czy_dobre x = abs_float (f x) <epsilon
  and popraw x = x -. (f x) /. (p x)
  in
    iteruj x popraw czy_dobre id;;
val newton : (float ->float) ->float ->float = <fun>

```

Zwróćmy uwagę na to, że procedura `popraw-newton` jest lokalna i ma dostęp do funkcji `f`.

Pod nazwą „metoda Newtona” znana jest też metoda liczenia pierwiastków kwadratowych. Jest to szczególny przypadek metody Newtona znajdowania zer, dla funkcji $f(x) = x^2 - a$. Funkcja ta ma zera w punktach $\pm\sqrt{a}$. Ponadto $f'(x) = 2x$, czyli nasz algorytm przekształca przybliżenie x w:

$$x - \frac{f(x)}{f'(x)} = x - \frac{x^2 - a}{2x} = \frac{2x^2 - x^2 + a}{2x} = \frac{x^2 + a}{2x} = \frac{x + \frac{a}{x}}{2}$$

czyli dokładnie tak, jak to ma miejsce w metodzie Newtona przybliżania pierwiastków kwadratowych. Zaczynając w punkcie 1 przybliżamy \sqrt{a} .

```

let sqrt a =
  let f x = a -. square x
  in newton f 1.;;
val sqrt : float ->float = <fun>

```

```

sqrt 17.64;;
- : float = 4.20000000000023643

```

Zauważmy jeszcze, że większość zdefiniowanych w tym przykładzie procedur to procedury wyższych rzędów.

6.7 Punkty stałe funkcji

Punkt stały funkcji f to taki x , że $f(x) = x$. W przypadku niektórych funkcji (i określonych x -ów) ciąg $x, f(x), f^2(x), f^3(x), \dots$ jest zbieżny do pewnego punktu stałego f . Jest tak np. jeżeli f jest przekształceniem zwięzającym.

Fakt: Jeżeli f jest funkcją ciągłą oraz ciąg $x, f(x), f^2(x), \dots$ jest zbieżny, to $\lim_{i \rightarrow \infty} f^i(x)$ jest punktem stałym f .

Możemy zaimplementować tę metodę przybliżania punktów stałych. Zastosujemy tutaj procedurę `iteruj` z poprzedniego przykładu:

```

let punkt_staly f x =
  let blisko x = abs_float (x -. f x) <epsilon
  in
    iteruj x f blisko f;;
val punkt_staly : (float ->float) ->float ->float = <fun>

```

Przykładową funkcją, której punkt stały można znaleźć tą metodą jest $\cos(x)$.

```
punkt_staly cos 1.0;;
0.73908...
```

Proszę to sprawdzić na jakimś nudnym wykładzie — wystarczy ciągle stukać w klawisz cos.

Obliczanie punktów stałych moglibyśmy zastosować do obliczania pierwiastków — punkt stały funkcji $y \rightarrow \frac{x}{y}$ to $\pm\sqrt{x}$. Jednak obliczenie:

```
punkt_staly (function y -> x /. y) 1.0;;
```

nie jest zbieżne:

$$1 \rightarrow \frac{x}{1} = x \rightarrow \frac{x}{x} = 1 \rightarrow \dots$$

W takich przypadkach czasami pomaga technika nazywana „wytłumieniem przez uśrednienie”. Uśrednienie funkcji f , to funkcja $x \mapsto \frac{f(x)+x}{2}$. Zauważmy, że dla dowolnej funkcji f , ma ona dokładnie takie same punkty stałe jak jej uśrednienie. Zamiast więc szukać punktów stałych f , możemy szukać punktów stałych uśrednienia f .

```
let usrednienie f =
  function x ->average x (f x);;
val usrednienie : (float ->float) ->float ->float = <fun>
```

```
let sqrt x =
  punkt_staly (usrednienie (function y ->x /. y)) 1.0;;
val sqrt : float ->float = <fun>
```

Jeśli zanalizujemy działanie powyższej procedury `sqrt`, to okaże się, że znowu uzyskaliśmy metodę pierwiastkowania Newtona.

6.8 Zastosowania procedur wyższych rzędów

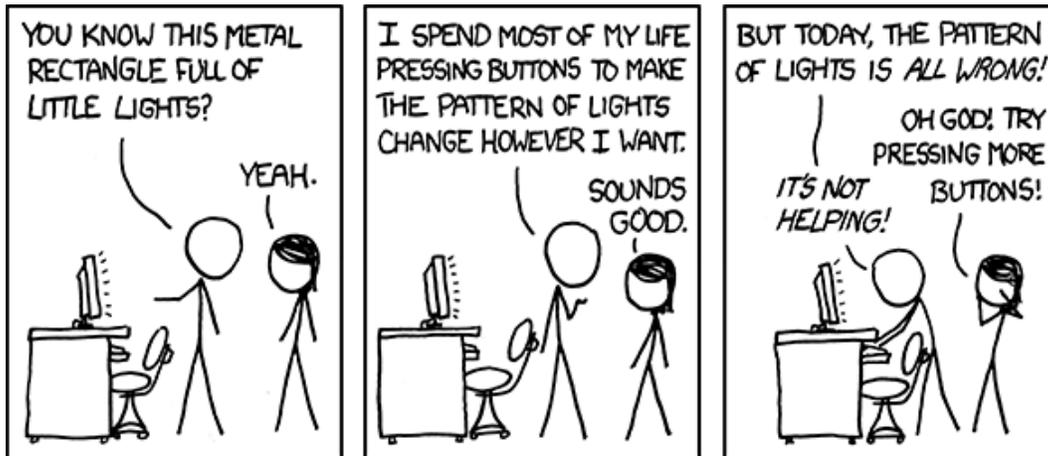
Procedury wyższych rzędów to jeden z tych elementów języka, który jest czysto funkcyjny. W językach imperatywnych procedury mogą być parametrami procedur, lecz nie wynikami. W przypadku języków funkcyjnych mamy pełną swobodę. Procedury mają tu te same prawa, co inne wartości.

Zastosowania procedur wyższych rzędów można podzielić na cztery grupy:

- Pewne pojęcia matematyczne, zwłaszcza te dotyczące funkcji, w naturalny sposób przekładają się na procedury wyższych rzędów, np.: sumy częściowe szeregów, składanie funkcji, różniczkowanie funkcji itp.
- Procedury są jeszcze jednym typem danych. Przyjmując takie podejście, procedury przetwarzające dane, jeśli te dane będą akurat procedurami, same będą procedurami wyższych rzędów.
- Procedury wyższych rzędów są również narzędziem abstrakcji. Jeżeli ten sam fragment kodu pojawia się w kilku miejscach, to naturalnym jest wyłonienie go w postaci (zwykłej) procedury. Jeżeli ten sam schemat kodu pojawia się w wielu miejscach, ale różni się wypełniającymi go fragmentami, to schemat ten możemy ująć w postaci procedury wyższego rzędu, a fragmenty do wypełnienia staną się parametrami proceduralnymi. Przykładem może być tu procedura `iteruj`.

- Zastosowanie standardowych procedur wyższych rzędów przetwarzających listy czy drzewa daje szczególnie zwarte definicje operacji na nich. Zobaczmy to w kolejnym wykładzie.

6.9 Deser



<http://xkcd.com/722/>

Ćwiczenia

1. Potęgowanie funkcji — wersja prostsza i szybsza, obie z dowodami poprawności. Zasymuluj ich działanie na prostych przykładach:

```
iterate 2 (function x ->x * (x+1)) 2
```

```
iterate 3 (function x ->x * (x+1)) 1
```

rozrysowując ramki. W przypadku szybszego potęgowania funkcji co tak na prawdę jest obliczane szybciej: funkcja wynikowa, czy jej wartość?

2. Niech $f : \mathcal{R} \rightarrow \mathcal{R}$ będzie funkcją:

- wzajemnie jednoznaczna, czyli 1-1 i „na”,
- ciągłą,
- rosnącą i to tak, że $\forall d > 0 f(x+d) - f(x) \geq d$, oraz
- taką, że $f(0) = 0$.

Zaimplementuj procedurę odwrotność, której wynikiem dla parametru f będzie przybliżenie f^{-1} z dokładnością zadaną przez stałą `epsilon` (czyli jeśli $g = (\text{odwrotność } f)$, to $\forall x |g(x) - f^{-1}(x)| \leq \text{epsilon}$).

3. Wygładzenie funkcji z odstępem dx polega na uśrednieniu $f(x - dx)$, $f(x)$ i $f(x + dx)$. Napisz procedurę wygładzającą daną funkcję z zadanym odstępem.
4. Zaimplementuj aproksymację funkcji za pomocą szeregu Taylora. Twoja procedura powinna mieć następujące parametry: liczbę sumowanych wyrazów szeregu, punkt, w którym badana jest przybliżana funkcja. Wynikiem powinno być przybliżenie funkcji. Zastosuj przybliżenie pochodnej oraz sumy częściowe szeregów, przedstawione na wykładzie.
5. [AS] Punktem stałym funkcji $y \rightarrow \frac{x}{y^{n-1}}$ jest $\sqrt[n]{x}$. Zaimplementuj obliczanie n -tego pierwiastka z x za pomocą obliczania punktu stałego i tłumienia przez uśrednianie. Wyznacz eksperymentalnie, ile razy należy stosować tłumienie w zależności od n . (Podpowiedź: k -krotne uśrednienie funkcji f , to funkcja postaci: `fun x -> $\frac{x \cdot (2^k - 1) + f(x)}{2^k}$.`)

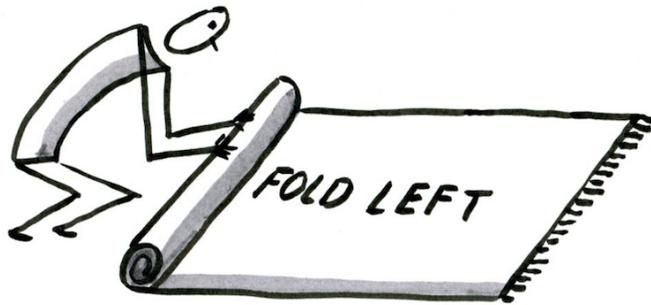
Wykład 7. Procedury wyższych rzędów i listy

Istnieje zestaw standardowych procedur wyższych rzędów specjalnie przeznaczonych do przetwarzania list. Większość procedur przetwarzających listy jest budowana według kilku powtarzających się schematów. Ujmując te schematy w postaci procedur wyższych rzędów, uzyskamy procedury wyższych rzędów, o których jest mowa.

Wszystkie przedstawione tu procedury wyższych rzędów są zdefiniowane w module `List`.

7.1 `fold_left`

Jeden z najczęściej występujących schematów procedur przetwarzających listy polega na tym, że przeglądamy kolejne elementy listy i obliczamy na ich podstawie pewien wynik. W trakcie przeglądania listy przechowujemy wynik pośredni, obliczony dla już przejranych elementów. W każdym kroku przeglądamy jeden element listy i uwzględniamy go w wyniku pośrednim. Możemy powiedzieć, że *kumulujemy* wpływ kolejnych elementów listy na wynik. Po przejrzaniu wszystkich elementów listy mamy gotowy wynik.



W schemacie tym mamy następujące elementy zmienne:

- wynik dla pustej listy,
- procedura *kumulująca* wpływ kolejnych elementów listy na wynik,
- listę do przetworzenia.

Procedura realizująca powyższy schemat, przeglądająca elementy zgodnie z ich kolejnością na liście, nosi tradycyjną nazwę `fold_left` i jest zdefiniowana następująco:

```
let rec fold_left f a l =
  match l with
  [] -> a |
  h::t -> fold_left f (f a h) t;;
val fold_left : ('a ->'b ->'a) ->'a ->'b list ->'a = <fun>
```

Pierwszym parametrem jest procedura kumulująca wynik. Ma ona dwa argumenty: dotychczas obliczony wynik i kolejny element listy do przetworzenia. Drugi argument to wynik dla pustej listy, a trzeci to lista do przetworzenia. Zauważmy, że mamy tu do czynienia z rekurencją ogonową.

Jej działanie można też przedstawić następującym wzorem:

$$\text{fold_left } f \ a \ [x_1; x_2; \dots; x_n] = f (\dots (f (f \ a \ x_1) \ x_2) \dots) \ x_n$$

Oto kilka prostych przykładów.

Przykład: Poniższe dwie procedury obliczają odpowiednio sumę i iloczyn wszystkich elementów listy:

```
let sum l = fold_left (+) 0 l;;
val sum : int list ->int = <fun>
```

```
let prod l = fold_left ( * ) 1 l;;
val prod : int list ->int = <fun>
```

Długość listy możemy również obliczyć używając `fold_left`. Wartości elementów listy nie mają tu znaczenia, tylko sama ich obecność.

```
let length l = fold_left (fun x _ ->x + 1) 0 l;;
val length : 'a list ->int = <fun>
```

Kumulowany wynik może oczywiście być czymś bardziej skomplikowanym niż liczbą. Oto implementacja procedury odwracającej listę `rev` za pomocą `fold_left`.

```
let rev l = fold_left (fun a h ->h::a) [] l;;
val rev : 'a list ->'a list = <fun>
```

Procedura `fold_left` ma też swoją wersję przeznaczoną do przetwarzania dwóch list tej samej długości:

```
let rec fold_left2 f a l1 l2 =
  match (l1,l2) with
  | ([],[]) ->a |
  | (h1::t1,h2::t2) ->fold_left2 f (f a h1 h2) t1 t2 |
  | _ ->failwith "Listy różnej długości";;
val fold_left2 : ('a ->'b ->'c ->'a) ->'a ->'b list ->'c list ->'a = <fun>
```

Przykład: Procedurę `fold_left2` możemy np. zastosować do obliczania iloczynu skalarnego wektorów reprezentowanych jako listy współrzędnych.

```
let iloczyn_skalarny l1 l2 =
  fold_left2 (fun a x y ->a + x * y) 0 l1 l2;;
val iloczyn_skalarny : int list ->int list ->int = <fun>
```

7.2 fold_right

Jak łatwo się domyślić, jeżeli istnieje procedura `fold_left`, to powinna być też procedura `fold_right`. I faktycznie jest. Różni się ona tym, że elementy listy są przeglądane w odwrotnej kolejności, niż występują na liście, czyli od prawej. Procedura ta jest zdefiniowana następująco:

```
let rec fold_right f l a =
  match l with
  [] -> a |
  h::t -> f h (fold_right f t a);;
val fold_right : ('a ->'b ->'b) ->'a list ->'b ->'b = <fun>
```

Pierwszym parametrem jest procedura kumulująca wynik. Ma ona dwa argumenty: kolejny element listy do przetworzenia i dotychczas obliczony wynik. Drugi argument to lista do przetworzenia, a trzeci to wynik dla pustej listy. Jej działanie można też przedstawić następującym wzorem:

$$\text{fold_right } f [x_1; x_2; \dots; x_n] a = f x_1 (\dots (f x_{n-1} (f x_n a)) \dots)$$

Zauważmy, że w definicji tej procedury nie występuje rekurencja ogonowa. Tak więc, jeżeli kolejność przetwarzania elementów listy jest bez znaczenia, to procedura `fold_left` może być bardziej efektywna.

Jednak w niektórych obliczeniach dużo prościej jest przetwarzać elementy w kolejności odwrotnej do tej, w jakiej występują na liście. Przykładem mogą być tu opisane dalej procedury `map` i `filter`. Oto kilka przykładów zastosowania procedury `fold_right`.

Przykład: Oto procedury z poprzedniego przykładu, zaimplementowane tym razem przy użyciu `fold_right`:

```
let sum l = fold_right (+) l 0;;
val sum : int list ->int = <fun>

let prod l = fold_right ( * ) l 1;;
val prod : int list ->int = <fun>

let length l = fold_right (fun _ x ->x + 1) l 0;;
val length : 'a list ->int = <fun>
```

Przykład: Przypomnijmy sobie jedno z ćwiczeń, polegające na napisaniu procedury `flatten`, która przekształca listę list elementów w listę elementów poprzez sklejanie list składowych. Procedurę tę można szczególnie zwięźle i elegancko zaimplementować używając `fold_right`.

```
let flatten l = fold_right (@) l [];;
val flatten : 'a list list ->'a list = <fun>

flatten [[1;2]; []; [3]; []; [4;5;6]];;
- : int list = [1; 2; 3; 4; 5; 6]
```

Dlaczego rozwiązanie tego zadania za pomocą `fold_left` byłoby mniej efektywne?

Procedury `fold_left` i `fold_right` są najbardziej elementarne spośród procedur, które przedstawiamy w tym punkcie. Porównując je pod tym kątem, okazuje się, że `fold_left` jest bardziej elementarna. Można zarówno zdefiniować `fold_right` za pomocą `fold_left`, jak i odwrotnie. Jednak definiując `fold_left` za pomocą `fold_right` tracimy ogonowość rekursji.

Żeby zdefiniować `fold_right` za pomocą `fold_left` wystarczy odwrócić daną listę (co można zrobić za pomocą `fold_left`) i przetwarzać elementy od lewej do prawej.

```
let fold_right f l a =
  let rev = fold_left (fun a h ->h::a) []
  in fold_left (fun x h ->f h x) a (rev l);;
val fold_right : ('a ->'b ->'b) ->'a list ->'b ->'b = <fun>
```

Definiując `fold_left` za pomocą `fold_right`, chcemy obliczyć:

$$\text{fold_left } f \ a \ [x_1; x_2; \dots; x_n] = f (\dots (f (f \ a \ x_1) \ x_2) \dots) \ x_n$$

Jednak musimy przetwarzać elementy w kolejności: x_n, x_{n-1}, \dots, x_1 . Pomysł polega na tym, aby kumulowane wartości były procedurami postaci:

```
fun x → x
fun x → f x xn
fun x → f (f x xn-1) xn
⋮
fun x → f (... (f x xi) ...) xn
⋮
fun x → f (... (f x x2) ...) xn
fun x → f (... (f (f x x1) x2) ...) xn
```

Procedury te przekształcają wynik dla początkowego fragmentu listy $[x_1; x_2; \dots; x_{i-1}]$ w wynik dla całej listy. Na koniec, wystarczy uzyskanej procedurze przekazać wynik dla pustej listy (a), a uzyskamy wynik dla całej listy.

```
let fold_left f a l =
  fold_right (fun h p ->function x ->p (f x h)) l (fun x ->x) a;;
val fold_left : ('a ->'b ->'a) ->'a ->'b list ->'a = <fun>
```

Podobnie jak `fold_left`, procedura `fold_right` również ma swój odpowiednik do przetwarzania dwóch list równocześnie.

```
let rec fold_right2 f l1 l2 a =
  match (l1,l2) with
  ([],[]) ->a |
  (h1::t1,h2::t2) ->f h1 h2 (fold_right2 f t1 t2 a) |
  - ->failwith "Listy różnej długości";;
val fold_right2 : ('a ->'b ->'c ->'c) ->'a list ->'b list ->'c ->'c = <fun>
```

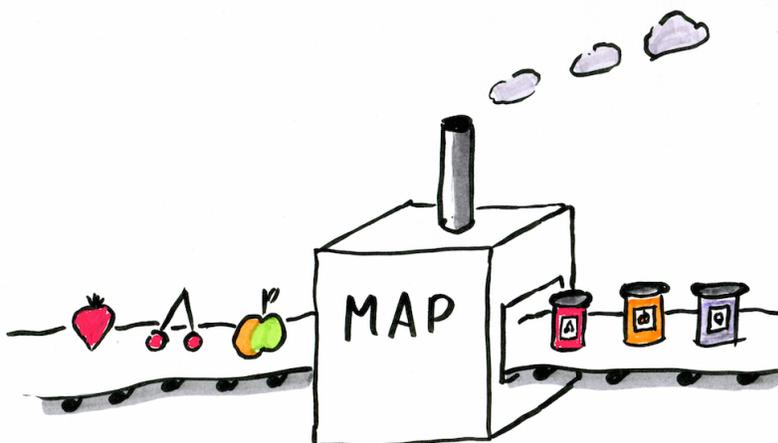
7.3 map

Kolejny często pojawiający się schemat procedur przetwarzających listy polega na tym, że do wszystkich elementów listy stosujemy to samo przekształcenie. Schemat ten został ujęty w postaci procedury `map`, która stosuje zadaną procedurę do wszystkich elementów danej listy i zwraca listę wyników.

$$\text{map } f [x_1; x_2; \dots; x_n] = [f x_1; f x_2; \dots; f x_n]$$

Oto implementacja tej procedury za pomocą `fold_right`:

```
let map f l = fold_right (fun h t ->(f h)::t) l [];;  
val map : ('a ->'b) ->'a list ->'b list = <fun>
```



Przykład: Przykłady użycia `map`:

```
map abs [6; -9; 4; -2; 0];;  
- : int list = [6; 9; 4; 2; 0]
```

```
map rev [[1;2]; []; [3]; []; [4;5;6]];;  
- : int list list = [[2; 1]; []; [3]; []; [6; 5; 4]]
```

Procedura `map` ma również swój odpowiednik do przetwarzania dwóch list tej samej długości:

```
let map2 f l1 l2 =  
  fold_right2 (fun h1 h2 t ->(f h1 h2)::t) l1 l2 [];;  
val map2 : ('a ->'b ->'c) ->'a list ->'b list ->'c list = <fun>
```

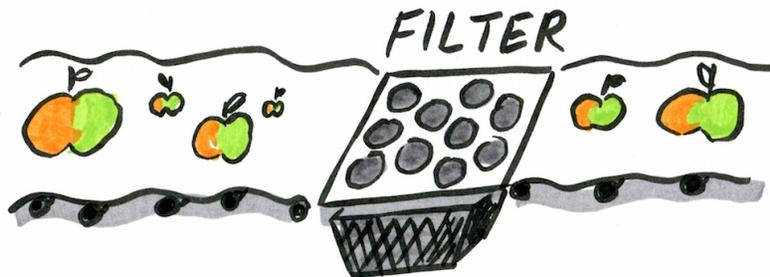
Przykład: Przykładem zastosowania procedury `map2` może być sumowanie wektorów reprezentowanych jako listy tej samej długości:

```
let suma_wektorow l1 l2 =
  map2 (+) l1 l2;;
val suma_wektorow : int list ->int list ->int list = <fun>
```

7.4 filter

Ostatni schemat procedur przetwarzających listy, jaki przedstawimy w tym wykładzie, to schemat procedury wybierającej z danej listy interesujące nas elementy. Sprawdzamy pewien warunek i zostawiamy tylko elementy spełniające go. Schemat ten realizuje procedura `filter`:

```
let filter p l =
  fold_right (fun h t ->if p h then h::t else t) l [];;
val filter : ('a ->bool) ->'a list ->'a list = <fun>
```



Procedury `filter` i `map` są, w pewnym sensie, dualne do siebie — `map` przekształca elementy, ale nie zmienia ich kolejności ani nie dokonuje żadnej selekcji, a `filter` nie zmienia wartości elementów, ale dokonuje ich selekcji.

Przykład: Procedury `filter` możemy użyć do zaimplementowania sita Eratostenesa.

```
let sito l =
  match l with
  [] ->[] |
  h::t ->filter (fun x ->x mod h <>0) t;;
val sito : int list ->int list = <fun>

let gen n =
  let rec pom acc k = if k <2 then acc else pom (k::acc) (k-1)
  in pom [] n;;
val gen : int ->int list = <fun>

let eratostenes n =
  let rec erat l = if l = [] then [] else (List.hd l)::(erat (sito l))
```

```

in erat (gen n);;
val eratostenes : int ->int list = <fun>

eratostenes 42;;
- : int list = [2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41]

```

7.5 Procedury wyższych rzędów przetwarzające drzewa

Nie tylko listy przetwarzamy rekurencyjnie — również drzewa są typową strukturą danych, jaką przetwarzamy rekurencyjnie. Drzewa dowolnego stopnia, w których węzłach przechowujemy wartości typu α , możemy zdefiniować następująco:

```

type  $\alpha$  tree = Node of  $\alpha$  *  $\alpha$  tree list

```

Drzewa sensowniej jest przetwarzać od liści do korzenia (ang. *bottom-up*), niż odwrotnie, bo wtedy mamy w korzeniu skumulowany jeden wynik. W pojedynczym kroku kumulujemy wartość z danego węzła i wartości obliczone dla jego poddrzew. Wyłania się z tego następujący schemat rekurencyjnego przetwarzania drzew:

```

let rec fold_tree f (Node (x,l)) =
  f x (map (fold_tree f) l);;
val fold_tree : ('a ->'b list ->'b) ->'a tree ->'b = <fun>

```



Przykład: Zdefiniujmy za pomocą `fold_tree` procedurę obliczającą wysokość drzewa. Wysokość drzewa to maksimum z wysokości poddrzew plus jeden. Przyjmujemy, że maksimum z pustej listy to zero.

```

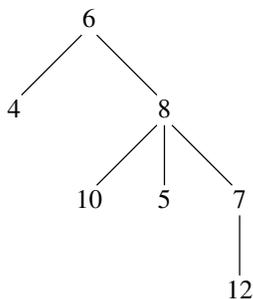
let height t =
  let maxl l = fold_left max 0 l
  in
  fold_tree (fun _ l ->maxl l + 1) t;;
val height : 'a tree ->int = <fun>

```

Zdefiniujemy odpowiedniki pozostałych procedur wyższych rzędów przetwarzających listy, dla drzew.

```
let map_tree f t =  
  fold_tree (fun x l ->Node (f x,l)) t;;  
val map_tree : ('a ->'b) ->'a tree ->'b tree = <fun>
```

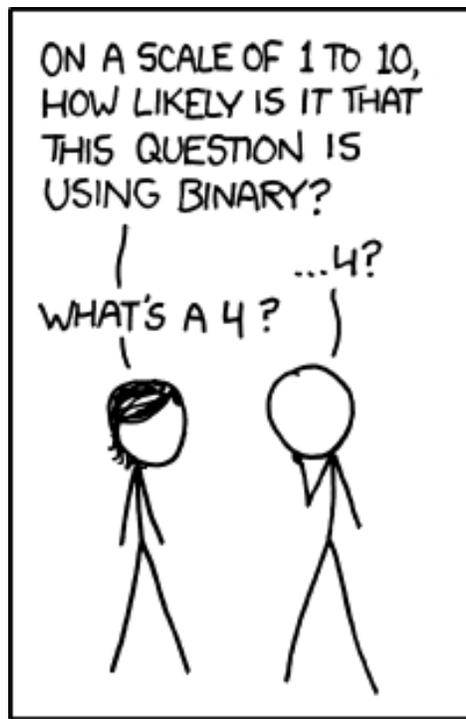
Przykład: Rozważamy drzewa typu `int tree`. Powiemy, że wartość w węźle drzewa jest *widoczna*, jeżeli na ścieżce od tego węzła do korzenia drzewa nie ma większej wartości (w sensie standardowego porządku $>$). W szczególności liczba w korzeniu drzewa jest zawsze widoczna, a liczby mniejsze od niej nie są nigdy widoczne. Chcemy napisać procedurę `widoczne: int tree → int`, która dla zadanego drzewa (zawierającego wyłącznie nieujemne liczby całkowite) wyznaczy liczbę widocznych liczb. Na przykład, dla drzewa przedstawionego na rysunku poprawnym wynikiem jest 4.



Zadanie to rozwiązywałoby się łatwiej, gdybyśmy mogli przetwarzać drzewo od korzenia do liści. Jednak `fold_tree` działa w przeciwnym kierunku. Chcąc użyć `fold_tree`, zamiast kumulować liczby powinniśmy kumulować procedury — dla danego poddrzewa obliczamy procedurę, która na podstawie maksimum na ścieżce od korzenia poddrzewa do korzenia całego drzewa oblicza liczbę widocznych elementów w poddrzewie.

```
let widoczne t =  
  let merge x l k =  
    if x <k then  
      fold_left (fun a h ->a + h k) 0 l  
    else  
      fold_left (fun a h ->a + h x) 0 l + 1  
  in  
  fold_tree merge t (-1);;  
val widoczne : int tree ->int = <fun>
```

7.6 Deser



<http://xkcd.com/953/>

Ćwiczenia

Ćwiczenia na listy (i nie tylko) z elementami procedur wyższych rzędów:

1. Napisz procedurę `exists`, która dla danego predykatu i listy sprawdzi, czy na liście jest element spełniający predykat. Wykorzystaj wyjątki tak, aby nie przeglądać listy, gdy to już nie jest potrzebne.
2. Napisz procedurę negującą predykat `non: ('a -> bool) -> ('a -> bool)`. Za pomocą tej procedury oraz procedury `exists` zdefiniuj procedurę `forall`, która sprawdza, czy dany predykat jest spełniony przez wszystkie elementy danej listy. Czy zastosowanie wyjątków w implementacji procedury `exists` nadal powoduje, że przeglądane są tylko niezbędne elementy listy?
3. Zapisz procedurę `append` za pomocą `fold_right/fold_left`.
4. Napisz procedurę obliczającą funkcję będącą sumą listy funkcji.
5. Napisz procedurę obliczającą funkcję będącą złożeniem listy funkcji.
6. Zapisz za pomocą `map` i `flatten` procedurę `heads`, której wynikiem dla danej listy list, jest lista pierwszych elementów niepustych list składowych. Puste listy składowe nie powinny wpływać na wynik.
7. Napisz procedurę obliczającą sumę elementów listy występujących po ostatniej liczbie ujemnej (lub wszystkich, jeżeli na liście nie ma liczb ujemnych).
8. Napisz procedurę `sumy: int list -> int list`, która dla danej listy $[x_1; \dots; x_n]$ oblicza listę postaci: $[x_1; x_1 + x_2; x_1 + x_2 + x_3; \dots, x_1 + x_2 + \dots + x_n]$.
Przykład: `sumy [1; 5; 2; 7; 12; 10; 5] = [1; 6; 8; 15; 27; 37; 42]`.
9. Napisz procedurę `codrugi: α list \rightarrow α list`, która z danej listy wybiera co drugi element (zaczynając od drugiego). Na przykład `codrugi [1; 2; 3; 4; 5] = [2; 4]`.
10. [PCh] Dany jest ciąg liczb całkowitych (niekoniecznie dodatnich). Element takiego ciągu nazwiemy *widocznym*, jeśli suma poprzedzających go elementów jest taka sama, jak suma elementów występujących po nim. Napisz procedurę `widoczne: int list \rightarrow int list`, która z danej listy liczb całkowitych wybierze te, które są widoczne.
11. Listę nazwiemy *ciekawą* jeśli żadne dwa kolejne jej elementy nie są sobie równe. Napisz procedurę `podzial: α list \rightarrow α list list`, która daną listę podzieli na minimalną liczbę spójnych fragmentów (zachowując ich kolejność), z których każdy jest ciekawy.
Na przykład, `podzial [3; 2; 2; 5; 7; 5; 4; 4; 3; 1] = [[3; 2]; [2; 5; 7; 5; 4]; [4; 3; 1]]`.
12. Dana jest lista liczb całkowitych. Element tej listy nazwiemy *prextremum* jeśli jest ostro większy lub ostro mniejszy od wszystkich elementów poprzedzających go na danej liście. Napisz procedurę `prextrema: int list \rightarrow int list`, która dla danej listy zwraca listę wszystkich jej prextremów. Elementy na liście wynikowej powinny być w takiej samej kolejności, w jakiej występowały na danej liście. Na przykład:
`prextrema [-2; 1; 0; 1; 3; 2; -1; 5; 4; -3; 2; 1; 7] = [-2; 1; 3; 5; -3; 7]`.

Rozwiązując to zadanie nie wolno Ci tworzyć żadnych własnych procedur rekurencyjnych. Zamiast tego należy wykorzystać standardowe procedury wyższych rzędów przetwarzające listy.

13. Napisz procedurę `wzrost : int list → int list`, która dla danej listy liczb całkowitych znajduje w niej najdłuższy spójny fragment ściśle rosnący i zwraca go. Na przykład:

```
wzrost [3; 4; 0; -1; 2; 3; 7; 6; 7; 8] = [-1; 2; 3; 7]
wzrost [] = []
```

Rozwiązując to zadanie nie wolno Ci tworzyć żadnych własnych procedur rekurencyjnych ani pętli. Możesz natomiast wykorzystać standardowe procedury przetwarzające listy poznane na wykładzie.

14. Dana jest lista $[x_1; x_2; \dots; x_n]$ reprezentująca kolejne chwile w czasie — tzn. x_i reprezentuje chwilę i . Dane są też dwa predykaty p i q . Powiemy, że w chwili i zachodzi „ p dopóki q ”, jeżeli istnieje takie $0 \leq k \leq n - i$, że zachodzi $p x_i, p x_{i+1}, \dots, p x_{i+k-1}$ oraz $q x_{i+k}$. Inaczej mówiąc, począwszy od chwili i musi zachodzić p , dopóki nie zajdzie q . W szczególności, jeżeli zachodzi $q x_i$, to p nie musi w ogóle zachodzić.

Napisz procedurę `dopoki : α list → ($\alpha \rightarrow$ bool) → ($\alpha \rightarrow$ bool) → int list` taką, że wynikiem `dopoki [x1; x2; ...; xn] p q` jest lista tych chwil w czasie, w których zachodzi p dopóki q .

Na przykład, dla następujących p i q :

i	1	2	3	4	5	6	7	8	9	10	11	12
$p x_i$	false	true	true	true	false	false	true	true	true	false	false	false
$q x_i$	false	false	true	false	true	false	false	true	false	false	true	false

poprawnym wynikiem jest: `[2; 3; 4; 5; 7; 8; 11]`.

Rozwiązując to zadanie nie wolno Ci tworzyć żadnych procedur rekurencyjnych ani konstrukcji imperatywnych. Możesz natomiast wykorzystać procedury wyższego rzędu przetwarzające listy, poznane na wykładzie.

15. Napisz procedurę `alternator : ($\alpha \rightarrow$ bool) → α list → α list`, która dla danego predykatu p i listy l :

- sprawdzi, które elementy listy spełniają predykat p ,
- pozostawi na danej liście tylko te elementy, które znajdują się między nieparzystymi i parzystymi z kolei elementami spełniającymi p (bez nich samych) .

Przykład:

```
alternator (fun x ->x mod 2 = 0) [5;2;3;5;7;4;3;1;8;6;3;0;1;2;7] = [3;5;7;1]
```

```
alternator (fun x ->x mod 2 = 0) [4;5;1;7;0;3;8;9] = [5;1;7]
```

16. Dla danej listy liczb całkowitych $l = [x_1; x_2; \dots; x_n]$, *minimum* (*maksimum*) nazwiemy taki fragment listy $x_i = x_{i+1} = \dots = x_j$, że:

- $1 \leq i \leq j \leq n$,
- jeśli $i > 1$, to $x_{i-1} > x_i$ ($x_{i-1} < x_i$),
- jeśli $j < n$, to $x_j < x_{j+1}$ ($x_j > x_{j+1}$).

Ekstremum oznacza minimum lub maksimum. Napisz procedurę `ekstrema : int list → int`, która dla danej listy policzy ile jest na niej ekstremów.

17. Napisz procedurę `prefiksy : int list → int list list`, która dla danej listy liczb całkowitych zwróci listę wszystkich jej prefiksów zakończonych zerem, w kolejności rosnącej ich długości. Na przykład:

`prefiksy [0;1;2;3;0;5;6;0;1] = [[0]; [0;1;2;3;0]; [0;1;2;3;0;5;6;0]]`

18. Napisz procedurę `prefiksy : int list → int list list`, która dla danej listy liczb całkowitych zwróci listę wszystkich jej prefiksów o dodatniej sumie elementów, uporządkowaną w kolejności rosnącej długości prefiksów. Na przykład:

`prefiksy [0; 2; 3; -7; -2; 4; 2; 1] =
[[0; 2]; [0; 2; 3]; [0; 2; 3; -7; -2; 4; 2]; [0; 2; 3; -7; -2; 4; 2; 1]]`

19. Maksymalne plateau³ w liście (bez rekurencji). Napisz procedurę `plateau : α list → int`, która oblicza długość najdłuższego stałego (spójnego) fragmentu danej listy.

20. Zdefiniuj, za pomocą `@` i `filter` uproszczoną wersję algorytmu quick-sort. W algorytmie tym elementy sortowanej listy dzielimy na nie większe i większe od pierwszego elementu listy, po czym obie listy wynikłe z podziału rekurencyjnie sortujemy.

21. Ciąg różnicowy ciągu $[x_1; x_2; \dots; x_n]$, to ciąg postaci $[x_2 - x_1; x_3 - x_2; \dots; x_n - x_{n-1}]$. Oblicz ciąg różnicowy zadanej listy liczb całkowitych.

22. Oblicz listę złożoną z pierwszych elementów kolejnych ciągów różnicowych danej listy, tzn.: głowy danej listy, głowy jej ciągu różnicowego, głowy ciągu różnicowego jej ciągu różnicowego itd.

23. Dana jest lista liczb zmiennopozycyjnych $[x_1; x_2; \dots; x_n]$. Jej uśrednienie, to lista postaci: $[\frac{x_1+x_2}{2.0}; \dots; \frac{x_{n-1}+x_n}{2.0}]$. Uśrednieniem listy jednoelementowej oraz pustej jest lista pusta.

Napisz procedurę `usrednienie : float list → float list`, która dla danej listy obliczy jej uśrednienie.

24. Dana jest lista $[x_1; \dots; x_n]$. Napisz procedurę `sumy : int list → int`, która znajduje maksymalną sumę postaci $x_i + x_{i+1} + \dots + x_j$. (Oczywiście dana lista może zawierać liczby ujemne.) Pusta suma (równa 0) jest również dopuszczalna.

25. Napisz funkcję `od_końca_do_końca : int list → int`, która dla danej niepustej listy $[a_1; \dots; a_n]$ obliczy $\min_{i=1,2,\dots,n} |a_i - a_{n+1-i}|$.

³Plateau to przedział, w którym funkcja jest stała (a jej wykres płaski), lub fragment ciągu, w którym jest on stały.

26. Załóżmy, że dana jest lista $[x_1; x_2; \dots; x_n]$. *Sufiksem* tej listy nazwiemy każdą listę, którą można uzyskać przez usunięcie pewnej liczby (od 0 do n) jej początkowych elementów. Tak więc sufiksami danej listy będzie n.p. ona sama, pusta lista, a także $[x_3; x_4; \dots; x_n]$.
Napisz (za pomocą `fold_left/fold_right`) procedurę `tails : α list \rightarrow α list list`, która dla danej listy tworzy listę wszystkich jej sufiksów, uporządkowaną wg malejących ich długości.
27. Rozważmy listę liczb całkowitych $[x_1; x_2; \dots; x_n]$. *Dołkiem* na tej liście nazwiemy taki indeks $1 < i < n$, że $x_i < \max(x_1, \dots, x_{i-1})$ oraz $x_i < \max(x_{i+1}, \dots, x_n)$. *Głębokość* takiego dołka to $\min(\max(x_1, \dots, x_{i-1}) - x_i, \max(x_{i+1}, \dots, x_n) - x_i)$. Napisz procedurę `dolek : int list \rightarrow int`, która dla danej listy liczb całkowitych wyznaczy maksymalną głębokość dołka na tej liście. Jeśli na danej liście nie ma żadnego dołka, to poprawnym wynikiem jest 0.
28. Napisz procedurę `pojedyncze : α list \rightarrow α list`, która dla danej listy wybierze z niej te elementy, które występują dokładnie raz z rzędu, np.:

`pojedyncze [1; 1; 3; 5; 5; 5; 3; 5] = [3; 3; 5]`

29. Lista trójek $[(l_1, s_1, r_1); \dots; (l_k, s_k, r_k)] : (\alpha \text{ list} * \text{int} * \text{int}) \text{ list}$ reprezentują listę elementów typu α w następujący sposób. Trójka (l, s, r) , gdzie $l = [x_1; x_2; \dots; x_s]$ jest s -elementową listą elementów typu α , reprezentuje ciąg r -elementowy, poprzez cykliczne powtarzanie elementów ciągu l :

$$\underbrace{x_1, x_2, \dots, x_s, x_1, x_2, \dots, x_s, x_1, \dots}_{r \text{ elementów}}$$

Z kolei lista trójek reprezentuje listę powstała w wyniku konkatenacji list odpowiadających poszczególnym trójkom.

Napisz procedurę `dekompresja : (α list * int * int) list \rightarrow int \rightarrow α` , która dla danej listy trójek i indeksu i wyznacza i -ty element listy wynikowej.

30. Rozważmy następującą metodę kompresji ciągów liczb całkowitych: Jeżeli w oryginalnym ciągu ta sama liczba powtarza się kilka razy z rzędu, to jej kolejne wystąpienia reprezentujemy za pomocą jednej tylko liczby. Konkretnie, i powtórzeń liczby k reprezentujemy w ciągu skompresowanym jako $2^{i-1} \cdot (2 \cdot k - 1)$.

Napisz procedurę `kompresuj : int list \rightarrow int list` kompresującą zadaną listę. Lista wynikowa powinna być oczywiście jak najkrótsza.

`kompresuj [1; 2; 2; 5; 11; 11; 2];;`

`- : int list = [1; 6; 9; 42; 3]`

31. Napisz procedurę `buduj_permutacje : α list list \rightarrow α \rightarrow α` , która przyjmuje permutację w postaci rozkładu na cykle i zwraca ją w postaci procedury.

Lista składowa postaci $[a_1; \dots; a_n]$, gdzie $a_i \neq a_j$ dla $1 \leq i < j \leq n$, reprezentuje cykl, czyli permutację przeprowadzającą a_i na $a_{(i \bmod n)+1}$ dla $1 \leq i \leq n$. Możesz założyć, że listy składowe są niepuste.

Lista list $[c_1; \dots; c_k]$, gdzie listy c_i dla $1 \leq i \leq k$ reprezentują cykle, reprezentuje permutację złożoną z tych cykli. Możesz założyć, że wszystkie cykle są rozłączne.

Przyjmujemy, że dla wartości x nie występujących na żadnej z list c_i mamy:

$$\text{buduj_permutacje } [c_0; \dots; c_k] \ x = x$$

W szczególności, przyjmujemy, że pusta lista reprezentuje permutację identycznościową.

```
let p = buduj_permutacje [[2; 1]; [8; 3; 4]; [5; 7; 6; 10]; [11]];;
```

```
map p [1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12];;
```

```
-. int list = [2; 1; 4; 8; 7; 10; 6; 3; 9; 5; 11; 12]
```

32. Napisz procedurę `podział : int list -> int list list`, która dla danej listy liczb całkowitych $l = [x_1; x_2; \dots; x_n]$ podzieli ją na listę list $[l_1; \dots; l_k]$, przy czym:

- $l = l_1 @ \dots @ l_k$,
- dla każdej listy l_i wszystkie elementy na takiej liście są tego samego znaku,
- k jest najmniejsze możliwe.

Przykład:

```
podział [1;3;0;-2;-2;-4;9] = [[1; 3]; [0]; [-2;-2;-4]; [9]]
```

33. Lista jest *monotoniczna* jeśli jest niemalejąca lub nierosnąca. Napisz procedurę `monotons : int list -> int`, która dla danej listy wyznaczy minimalną liczbę spójnych fragmentów, na które trzeba ją podzielić, tak żeby każdy fragment był monotoniczny. (Dla pustej listy poprawnym wynikiem jest zero.) Na przykład, `monotons [2; 3; 3; 5; 0; -1; -7; 1; 3; 5; 5; 2; 4; 6] = 4`. Daną listę można podzielić tak: `[2; 3; 3]`, `[5; 0; -1; -7]`, `[1; 3; 5; 5]`, `[2; 4; 6]`.

34. Napisz procedurę `podział : int list -> int list list`, która dla danej listy liczb całkowitych $l = [x_1; x_2; \dots; x_n]$ podzieli ją na listę list $[l_1; \dots; l_k]$, przy czym:

- $l = l_1 @ \dots @ l_k$,
- każda z list l_i jest ściśle rosnąca,
- k jest najmniejsze możliwe.

Przykład:

podział $[1;3;0;-2;-2;4;9] = [[1; 3]; [0]; [-2]; [-2;4;9]]$

35. Zadeklaruj typ danych reprezentujący abstrakcyjną składnię wyrażeń arytmetycznych. Napisz procedurę obliczającą wartość wyrażenia.

Rozszerz składnię wyrażeń o zmienne. Procedura obliczająca wartość wyrażenia będzie wymagać dodatkowego parametru — wartościowania zmiennych, czyli funkcji, która nazwie zmiennej przyporządkowuje jej wartość.

36. Dane są: definicja typu `tree` i procedura `fold_tree`:

```
type 'a tree = Node of 'a * 'a tree list;;
```

```
let rec fold_tree f (Node (x,l)) =
```

```
  f x (map (fold_tree f) l);;
```

Użyj procedury `fold_tree` do zaimplementowania:

- policzenia liczby węzłów w drzewie,
- sprawdzenia czy drzewo jest zrównoważone, tzn. wszystkie jego liście są na tej samej głębokości,
- sprawdzenia czy drzewo regularne, tzn. wszystkie jego wierzchołki, z wyjątkiem liści, są tego samego stopnia,
- procedury `preorder/postorder` : `'a tree -> 'a list`, która przekształca dane drzewo w listę jego elementów w porządku pre-order/post-order (zwróć uwagę na efektywność),

37. Dane są: definicja typu `bin_tree` i procedura `fold_bin_tree`:

```
type 'a bin_tree = Node of 'a bin_tree * 'a * 'a bin_tree | Null;;
```

```
let rec fold_bin_tree f a t =
```

```
  match t with
```

```
    Null ->a |
```

```
    Node (l,x,r) ->f x (fold_bin_tree f a l) (fold_bin_tree f a r);;
```

Użyj procedury `fold_bin_tree` do zaimplementowania:

- (a) Policzenia liczby węzłów w drzewie.
- (b) Policzenia wysokości drzewa.
- (c) Policzenia średnicy drzewa.

- (d) Sprawdzenia czy drzewo jest drzewem BST (dla drzewa liczb zmiennopozycyjnych, tzn. typu `float bin_tree`).
- (e) Sprawdzenia, czy drzewo ma kształt drzewa AVL (tzn., czy dla każdego węzła drzewa wysokości jego obu poddrzew różnią się o co najwyżej 1).
- (f) Listy list węzłów drzewa znajdujących się na kolejnych poziomach głębokości.
- (g) Zaimplementowania procedury `map_bin_tree` — odpowiednika procedury `map_tree` dla drzew binarnych.
- (h) Skonstruowania listy elementów znajdujących się w wierzchołkach drzewa, w kolejności infiksowej.
- (i) Procedury `path : α tree \rightarrow α list`, która znajduje najdłuższą ścieżkę (jedną z) od korzenia do liścia i zwraca listę wartości znajdujących się w węzłach tworzących tę ścieżkę, w kolejności od korzenia do liścia.
- (j) Zaimplementowania procedury `levels : α tree \rightarrow α list list`, która dla danego drzewa zwraca listę list węzłów drzewa znajdujących się na kolejnych poziomach głębokości. Listy składowe powinny być podane w kolejności od korzenia w głąb drzewa. Wartości z węzłów znajdujących się na tej samej głębokości powinny być uporządkowane od lewej do prawej.

Wytyczne dla prowadzących ćwiczenia

To duży zestaw zadań. Należy na niego poświęcić ok. 2 zajęć. Zamiast procedur rekurencyjnych należy stosować standardowe procedury wyższych rzędów przetwarzające listy. W zadaniach wymagających użycia `fold`-ów zwracamy uwagę, której procedury należy użyć: `fold_left`, `fold_right`, czy też nie ma to znaczenia. Dopóki na wykładzie nie będzie mowy o rzędach funkcji i analizie złożoności, nieformalnie analizujemy, czy rekurencja ogonowa w `fold_left` daje stały koszt pamięciowy, czy też i tak musi on być liniowy. Po omówieniu analizy złożoności na wykładzie analizujemy złożoność rozwiązań.

Nie wszystkie zadania zdąży się zrobić — pozostałe studenci mogą rozwiązać sami w ramach przygotowania do kolokwium. Warto jednak zrobić zadanie 37 dotyczące przetwarzania drzew.

Ad. 20 Należy zwrócić uwagę na przypadki brzegowe, w szczególności na to, czy rekurencja może się zapętlić.

Ad. 37 Należy zwrócić uwagę na złożoność czasową. Uzyskanie liniowej złożoności może wymagać kumulowania za pomocą `fold_tree` nie wyniku, ale odpowiedniej procedury konstruującej wynik.

Wykład 8. Model obliczeń

W tym wykładzie przedstawimy uproszczoną semantykę operacyjną poznanego fragmentu Ocaml. Z jednej strony będzie to model wykonywania obliczeń na tyle dokładny, że pozwoli nam określić wyniki, a później również złożoność czasową i pamięciową obliczeń. Z drugiej strony będzie on uproszczony, gdyż pewne kwestie pominiemy, np. kontrolę typów, rozmaite optymalizacje wykonywane przez kompilator, czy sposób realizacji tych elementów języka, których jeszcze nie poznaliśmy. Wyjaśnimy też pewne szczegóły wykonywania obliczeń, które pominiemy wcześniej.

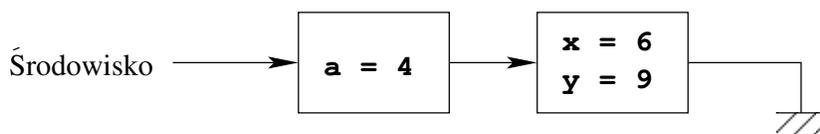
8.1 Środowisko i ramki.

Przedstawiając podstawy Ocaml mówiliśmy o *środowisku* przypisującym nazwom stałe ich wartości. Dodatkowo takich środowisk może być wiele: środowisko zawierające wszystkie aktualnie globalnie zdefiniowane stałe, środowiska powstające na potrzeby definicji lokalnych i wywołań procedur.

Środowisko jest zrealizowane jako lista jednokierunkowa, tzw. *lista ramek*. Każda ramka zawiera identyfikator (lub identyfikatory) i ich wartości (lub wskaźniki do takich wartości) oraz wskaźnik do kolejnej ramki tworzącej środowisko. W przypadku wartości prostych typów wbudowanych (takich, które można zmieścić w jednym słowie maszynowym), są one bezpośrednio pamiętane w ramce. W przypadku wartości złożonych, w ramce znajduje się wskaźnik do wartości.

Poszukując w środowisku wartości stałej, przeglądamy kolejne ramki środowiska. Pierwsza ramka, która zawiera nazwę stałej, określa jej wartość.

Dalej będziemy utożsamiać środowisko ze wskaźnikiem do pierwszej ramki tworzącej środowisko. Środowisko reprezentowane przez listę ramek bez pierwszej ramki będziemy nazywać *środowiskiem otaczającym*, a wskaźnik prowadzący z jednej ramki do kolejnej będziemy nazywać *wskaźnikiem do środowiska otaczającego*.

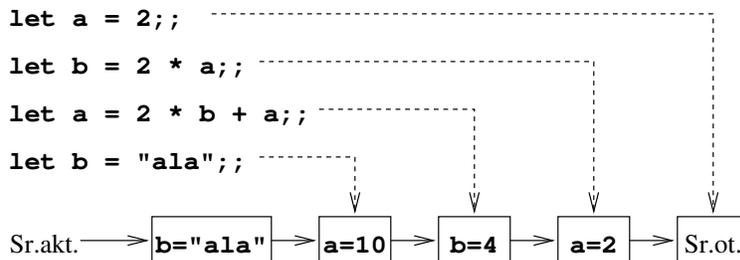


8.2 Definicje globalne

Prowadząc interakcję z kompilatorem, pracujemy w kontekście tzw. *aktualnego środowiska (globalnego)*. Każda dodana definicja zmienia to środowisko. Zdefiniowanie nowych wartości powoduje dodanie nowej ramki zawierającej zdefiniowane wartości. Dotychczasowe środowisko staje się środowiskiem otaczającym tej ramki. Utworzona ramka staje się nowym aktualnym środowiskiem (globalnym).

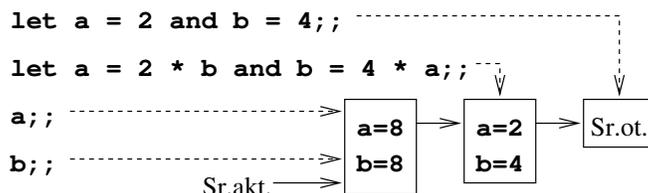
Mechanizm ten pozwala na przesłanianie istniejących w środowisku stałych. Jeśli zdefiniujemy nową stałą o takiej samej nazwie, to będzie ona znajdowała się we wcześniejszej ramce i to właśnie ona będzie znajdowana w aktualnym środowisku. Jednak poprzednio zdefiniowana stała nie jest usuwana ze środowiska. Jak zobaczymy, będzie to miało znaczenie przy obliczaniu procedur odwołujących się do wcześniej zdefiniowanych stałych.

Przykład:



Ramka może zawierać więcej niż jedną stałą, jeżeli w definicji użyjemy spójnika `and`.

Przykład:



8.3 Wartości typów danych

Wartości prostych typów wbudowanych (takie jak `int`, `bool`, czy `float`) są pamiętane w ramkach wprost. Wartości typów złożonych są pamiętane jako wskaźniki do odpowiednich struktur wskaźnikowych reprezentujących te wartości. Dzięki przyjęciu takiej konwencji, reprezentacja dowolnej wartości zajmuje zawsze w ramce stałą pamięć. (Dotyczy to zarówno wartości przechowywanych w ramkach, jak i przekazywanych obliczanych wartości wyrażeń.) W rezultacie definiując stałe o złożonych wartościach, nie kopiujemy tych wartości, a jedynie wskaźniki do nich. Dodatkowo, ponieważ raz powstałe wartości nie ulegają zmianie, struktury danych reprezentujące różne wartości mogą współdzielić pewne fragmenty.

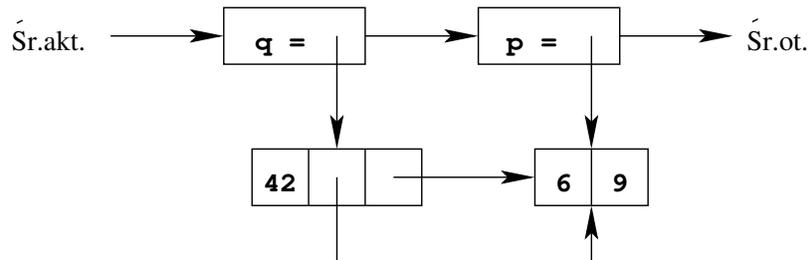
Wartości różnych złożonych typów danych są generalnie reprezentowane jako rekordy złożone ze wskaźników do wartości składowych:

- Element produktu kartezjańskiego jest reprezentowany jako n -tka wartości lub wskaźników do tworzących go wartości.
- Rekord — podobnie jak produkt kartezjański jest reprezentowany jako rekordy złożony z wskaźników do wartości pól.
- Warianty bez argumentów są pamiętane jako stałe wyznaczające operator, a warianty z argumentami są pamiętane jako pary: stała wyznaczająca operator i wskaźnik do argumentu.
- Lista pusta jest reprezentowana jako stała, a lista niepusta jest reprezentowana jako para: głowa i ogon.

```

let p = (6, 9);;
let q = (42, p, p);;

```



8.4 Wartości proceduralne

Na wartość procedury składają się trzy elementy:

- nazwy parametrów formalnych procedury,
- treść procedury,
- wskaźnik do środowiska, w którym zdefiniowano procedurę.

Pierwsze dwa elementy przekładają się w trakcie kompilacji na kod procedury. Natomiast wskaźnik do środowiska, w którym zdefiniowano procedurę, może być ustalony dopiero w trakcie obliczeń — dotyczy to np. procedur lokalnych. Reasumując, wartość procedury jest reprezentowana jako para wskaźników: do kodu skompilowanej procedury i do środowiska, w którym procedura została zdefiniowana. W przypadku procedur rekurencyjnych „środowisko, w którym zdefiniowano procedurę” zawiera jej własną definicję.

Uwaga: Formalnie rzecz biorąc, każda procedura ma tylko jeden argument. Jednak w sytuacjach, gdy takie uproszczenie nie będzie prowadzić do nieporozumień, będziemy dopuszczać obiekty proceduralne z wieloma argumentami. Uproszczenie takie jest możliwe wtedy, gdy w zastosowaniach procedury wszystkie jej parametry otrzymują wartości.

8.4.1 Zastosowanie procedury do argumentów

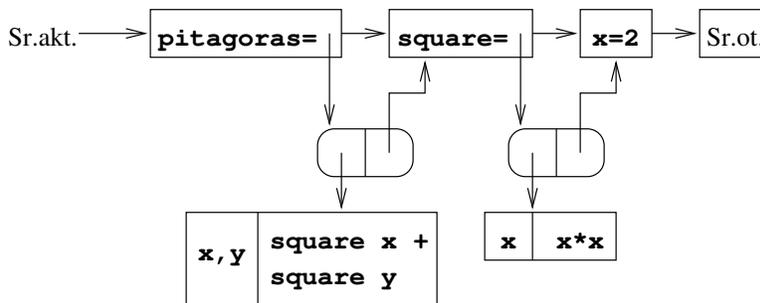
Zastosowanie procedury, czyli obliczenie jej wartości, polega na:

- wyliczeniu wartości wszystkich potrzebnych elementów: procedury i jej argumentów,
- stworzeniu nowej ramki, dla której środowiskiem otaczającym jest środowisko, w którym zdefiniowano procedurę, a nazwom parametrów formalnych przyporządkowano wartości argumentów,
- wyliczeniu w takim środowisku wartości wyrażenia stanowiącego treść procedury.

Przykład: Wprowadzenie definicji:

```
let x = 2;;
let square x = x * x;;
let pitagoras a b = square a + square b;;
```

powoduje następujące rozszerzenie środowiska:

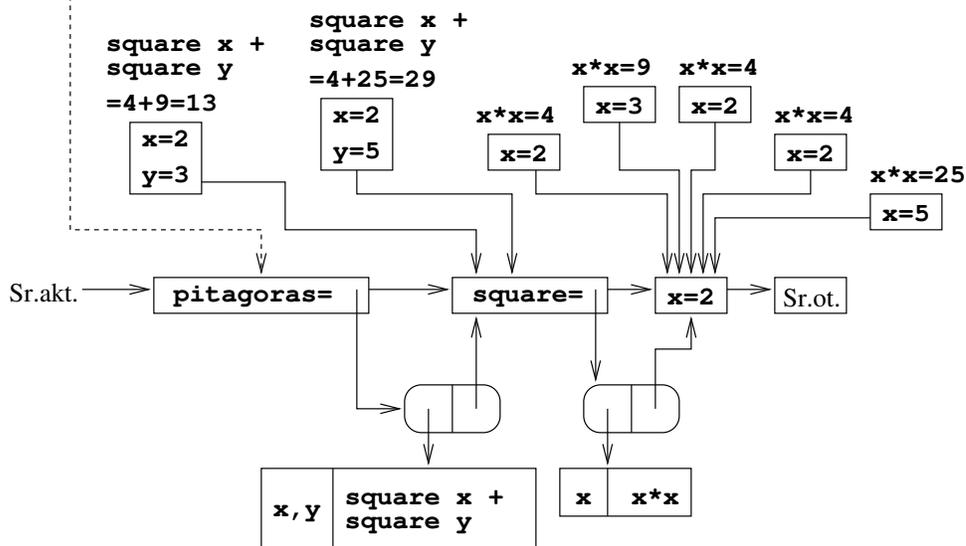


Obliczenie wyrażenia:

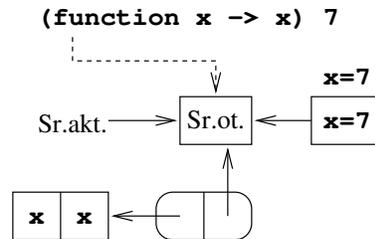
```
pitagoras x (x + 1) + pitagoras x ((square x) + 1);;
```

będzie przebiegać następująco:

```
pitagoras x (x+1) + pitagoras x ((square x) + 1) =
pitagoras 2 3 + pitagoras 2 ((square 2) + 1) =
13 + pitagoras 2 ((square 2) + 1) =
13 + pitagoras 2 5 =
13 + 29 =
42
```

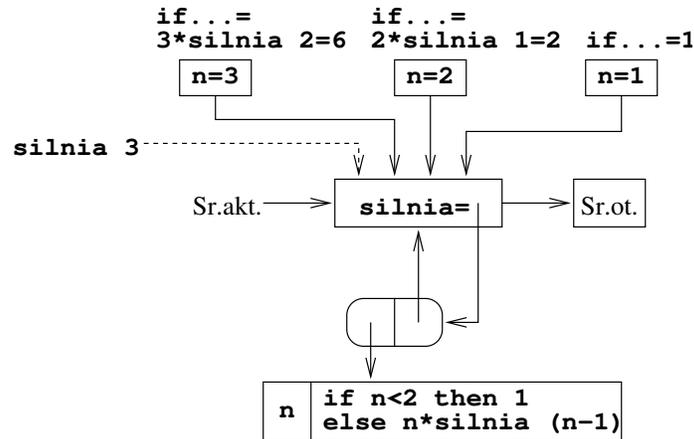


Przykład: Prześledźmy obliczenie prostego wyrażenia zawierającego λ -abstrakcję:



Przykład: Zastosowanie procedury rekurencyjnej:

```
let rec silnia n =
  if n <2 then 1 else n * silnia (n - 1);;
silnia 3;;
```



8.4.2 Rekurencja ogonowa

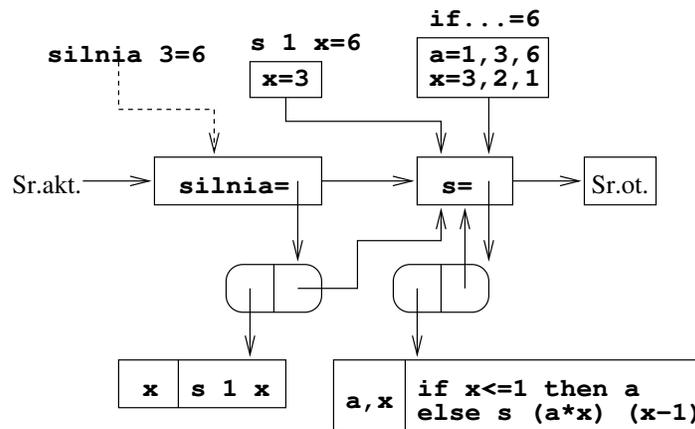
Jeśli wartości zwracane przez wszystkie wywołania rekurencyjne, bez żadnego przetwarzania, są przekazywane jako wynik danej procedury, to mówimy o rekurencji ogonowej. W przypadku rekurencji ogonowej, w momencie wywołania rekurencyjnego nie musimy tworzyć nowej ramki — możemy wykorzystać istniejącą ramkę, zmieniając jedynie pamiętane w niej wartości argumentów. W przypadku rekurencji ogonowej koszt pamięciowy związany z ramkami dla kolejnych wywołań rekurencyjnych jest stały, gdyż jest to tylko jedna ramka.

Przykład: Oto procedura obliczająca silnię z rekurencją ogonową. Zwróćmy uwagę na dodatkowy parametr *a*, w którym kumuluje się wynik. Takie dodatkowe parametry są nazywane *akumulatorami*. Dzięki zastosowaniu akumulatora mamy rekurencję ogonową.

```

let rec silnia_pom a x =
  if x <= 1 then a else silnia_pom (a * x) (x - 1);;
let silnia x = silnia_pom 1 x;;
silnia 3;;

```

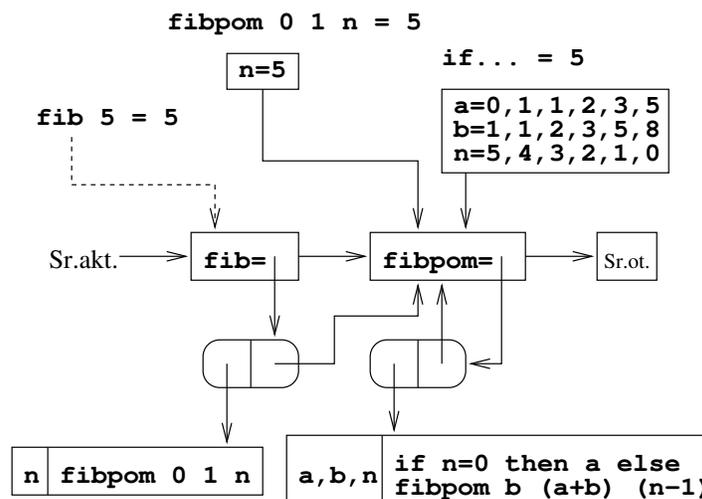


Przykład: Oto procedura obliczająca liczby Fibonacciego z rekurencją ogonową i akumulatorem.

```

let rec fib_pom a b n =
  if n = 0 then a else fib_pom b (a + b) (n - 1);;
let fib n = fib_pom 0 1 n;;
fib 5;;

```



8.4.3 Definicje lokalne

Mówiliśmy wcześniej, że wyrażenie postaci:

```
let a = b in c
```

jest podobne do zastosowania λ -abstrakcji postaci:

```
(function a ->c) b
```

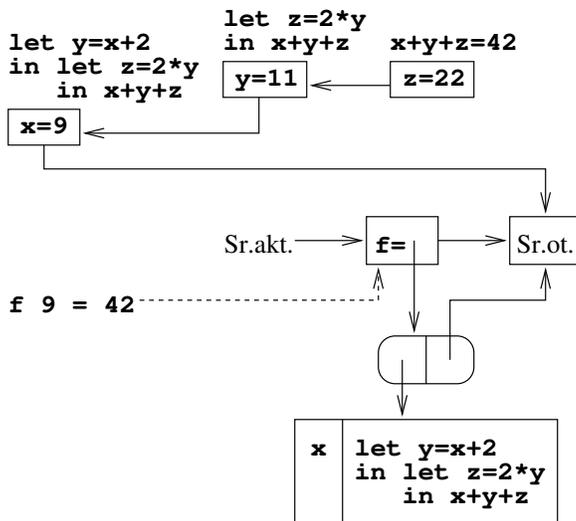
Sposób obliczania wyrażenia `let ...in ...` jest analogiczny do sposobu obliczania takiej λ -abstrakcji:

- tworzona jest ramka zawierająca definicję symboli lokalnych (otaczającym ją środowiskiem jest aktualne środowisko),
- w tak powstałym środowisku wyliczana jest wartość wyrażenia stojącego po `in`,
- po czym przywracane jest aktualne środowisko sprzed obliczenia całego wyrażenia.

Jeżeli definicje lokalne są pozagnieżdżane, to powstaje odpowiednio więcej ramek z wartościami symboli lokalnych.

Przykład: Następujący przykład pokazuje sposób realizacji definicji lokalnych.

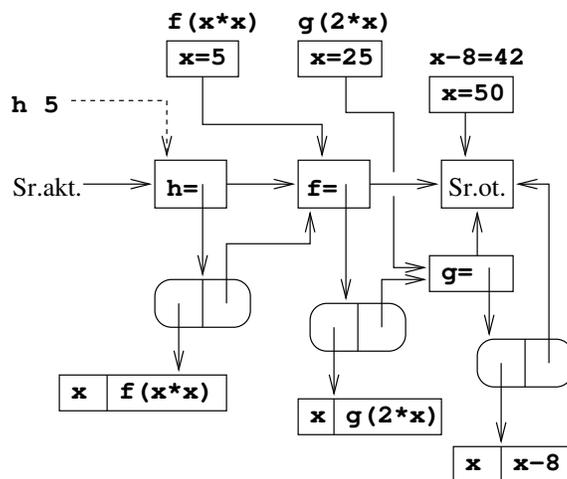
```
let f x =
  let y = x + 2
  in
    let z = 2 * y
    in
      x + y + z;;
f 9;;
```



Przykład: Niniejszy przykład ilustruje mniej typową kombinację użycia definicji lokalnych i procedur.

```
let f =
  let g x = x - 8
  in
    fun x ->g (2 * x);;
let h x = f (x * x);;
h 5;;
```

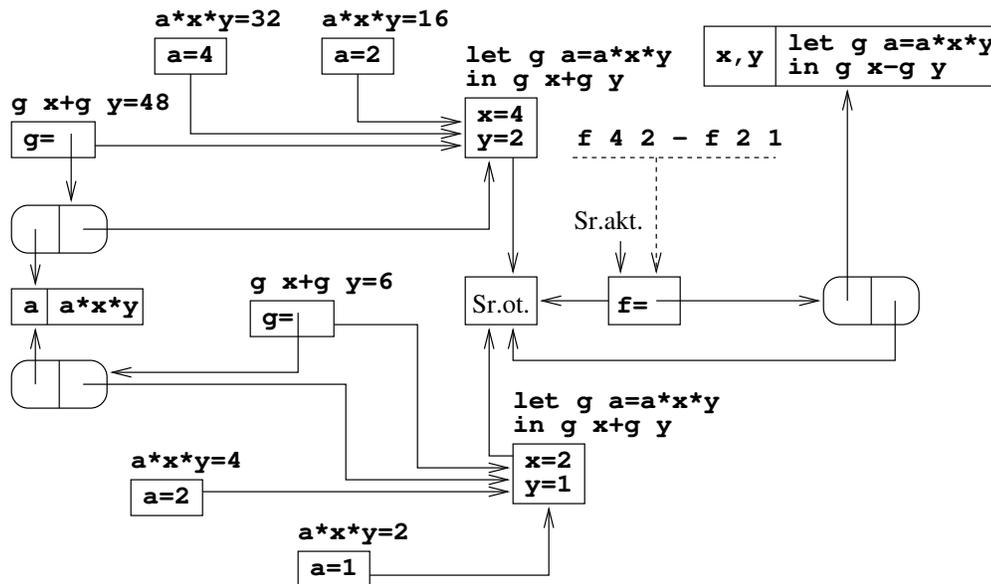
Zwróćmy uwagę na procedurę pomocniczą g. Procedura ta jest widoczna tylko wewnątrz procedury f, a jednak jej definicja ma tylko jedną instancję.



Przykład:

```
let f x y =
  let
    g a = a * x * y
  in
    g x + g y;;
f 4 2 - f 2 1;;
```

W tym przykładzie, w każdym wywołaniu procedury `f` powstaje osobna instancja procedury lokalnej `g`. Jest to naturalne zważywszy, że procedura `g` korzysta z argumentów wywołania procedury `f`.



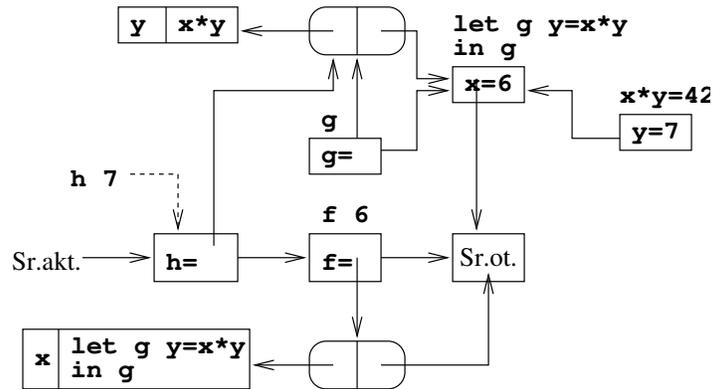
8.5 Ramki a rekordy aktywacji

Ramkom odpowiadają w językach imperatywnych *rekordy aktywacji*. Rekord aktywacji to porcja informacji towarzysząca pojedynczemu wywołaniu procedury. W rekordach aktywacji pamiętane są np. wartości zmiennych lokalnych. Rekordy aktywacji są pamiętane na stosie. W momencie powrotu z wywołania procedury, związany z nim rekord aktywacji jest niszczone.

Pokażemy na przykładach, że w naszym modelu obliczeniowym tak nie jest. Wynika to z funkcyjnego charakteru języka oraz z tego, że procedury mogą być wynikami procedur. Ramka, zawierająca lokalne definicje lub argumenty procedury, nie musi stawać się od razu zbędna, gdyż mogą na nią wskazywać wartości proceduralne. Jeśli spojrzymy na wszystkie ramki, to nie tworzą one listy ani stosu, ale drzewo (wskazniki na otaczające środowisko prowadzą w górę tego drzewa). Ramki, które trzeba trzymać w pamięci, bo są potrzebne, też nie muszą tworzyć listy, tylko drzewo. Zjawisko to występuje np. wówczas, gdy wynikiem procedury jest procedura mająca dostęp do lokalnie zdefiniowanych symboli.

Przykład: Niniejszy przykład pokazuje sytuację, gdy ramki powstałe w czasie wywołania procedury mogą być potrzebne po jego zakończeniu.

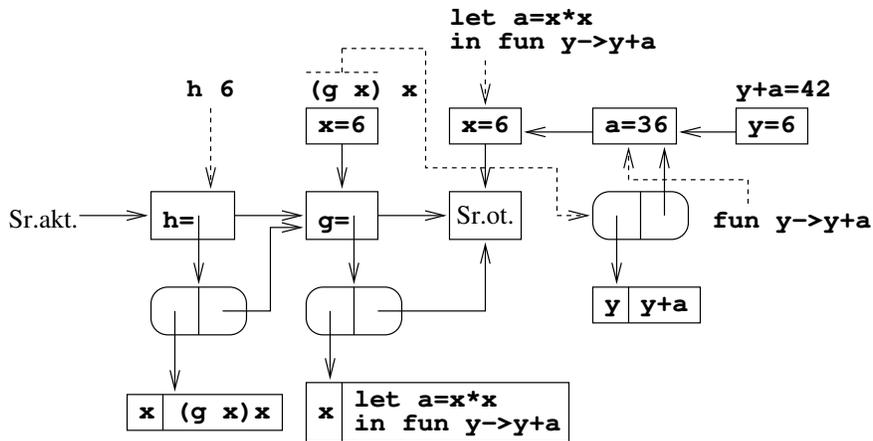
```
let f x =
  let g y = x * y
  in g;;
let h = f 6;;
h 7;;
```



Ramka zawierająca definicję procedury `g` powstaje w wyniku wywołania `f 6`, ale jest potrzebna później, do obliczenia `h 7`.

Przykład: Poniższy przykład ilustruje sposób przekazywania wielu argumentów, tak jak to jest realizowane — po jednym na raz.

```
let g x =
  let
    a = x * x
  in
    fun y -> y + a;;
let h x = (g x) x;;
h 6;;
```



8.6 Odśmiecianie

W przedstawionym modelu obliczeń cały czas są tworzone ramki i elementy struktur danych. Co więcej, jak widzieliśmy, nie możemy usuwać ramek powstałych na potrzeby definicji lokalnych i zastosować procedur po obliczeniu wyników, gdyż mogą one być nadal potrzebne. Gdy

brakuje wolnej pamięci, uruchamiany jest proces *odśmiecania*. Proces ten usuwa wszystkie te ramki i elementy struktur danych, które nie są dostępne i tym samym nie mogą być już do niczego potrzebne.

W tym celu przeszukuje się strukturę wskaźnikową ramek i danych. Za punkt wyjścia służą wszystkie te środowiska, w kontekście których są aktualnie obliczane wyrażenia, plus aktualne środowisko globalne. Wszystkie te ramki i rekordy, które są z nich dostępne (pośrednio lub bezpośrednio) są potrzebne. Natomiast te, które są niedostępne, mogą być usunięte.

Obliczając złożoność czasową programów, możemy pominąć koszt odśmiecania. Zwykle odśmiecanie jest tak zrealizowane, że jego koszt zamortyzowany jest stały. Można go wliczyć w koszt czasowy tworzenia nowych ramek i rekordów, obciążając utworzenie każdej ramki i rekordu stałym kosztem czasowym.

Obliczanie kosztu pamięciowego jest skomplikowane. Należy prześledzić płataninę powstających ramek i rekordów, sprawdzić, które są w danym momencie niezbędne, a które mogą zostać odśmieczone i określić minimalną wielkość pamięci niezbędnej do przeprowadzenia obliczeń. Dodatkowo możemy przyjąć konwencję, że rozmiar danych nie wlicza się do złożoności pamięciowej. Wynika to z charakteru programowania funkcyjnego. Przekazane dane nie mogą być modyfikowane przez program.

8.7 Rekurencja ogonowa

Przykład: Ta definicja wygląda na ogonową, jednak ...

```
let rec f a n x =  
  if n=0 then x  
  else  
    let g x = a x  
    in  
      f g (n-1) (g x);;  
f (fun x ->x * (x+1)) 3 1;;
```

[[TODO: rysunek]]

Dodatkowe ograniczenie na to kiedy możemy stosować rekurencję ogonową: Jeśli do ramki prowadzą (potrzebne) wskaźniki, to nie możemy zmienić jej zawartości.

8.8 Deser



<http://xkcd.com/138/>

Wykład 9. Analiza kosztów

9.1 Rzędy funkcji

$$f(x) = \Theta(g(x)) \Leftrightarrow \exists_{c_1, c_2 \in \text{real}, c_1, c_2 > 0, n_0 \in \text{nat}} \forall_{n \in \text{nat}, n \geq n_0} c_1 g(n) \leq f(n) \leq c_2 g(n)$$

$$f(x) = O(g(x)) \Leftrightarrow \exists_{c \in \text{real}, c > 0, n_0 \in \text{nat}} \forall_{n \in \text{nat}, n \geq n_0} 0 \leq f(n) \leq cg(n)$$

$$f(x) = \Omega(g(x)) \Leftrightarrow \exists_{c \in \text{real}, c > 0, n_0 \in \text{nat}} \forall_{n \in \text{nat}, n \geq n_0} 0 \leq cg(n) \leq f(n)$$

Dla funkcji nieujemnych zachodzą ponadto następujące fakty:

$$f(x) = \Theta(g(x)) \Leftrightarrow f(x) = \Omega(g(x)) \wedge f(x) = O(g(x))$$

$$f(x) = \Omega(g(x)) \Leftrightarrow g(x) = O(f(x))$$

Przykłady: Jak się mają do siebie funkcje:

- $n^2 + 100\,000$,
- $0.000001n^{\frac{1}{2}}$,
- 4 ,
- 2^{2n} ,
- $\log_2 n$,
- $2\sqrt{4n}$,
- n^n ,
- $\ln(10n)$,
- 4^n ,
- $n!$,
- 10^n ,
- 0 .

9.2 Przedstawienie kosztu jako funkcji

Koszt, to ilość (określonego) zasobu potrzebnego do wyliczenia wyniku. Ilość ta zależy od konkretnych danych. Jeśli oznaczymy przez D zbiór możliwych danych, to liczbę potrzebnych zasobów możemy określić funkcją $\varphi : D \rightarrow N$.

Zwykle interesuje nas ilość potrzebnych zasobów w zależności od określonego aspektu danych, np.:

- rozmiaru danych,
- rozmiaru macierzy,
- jeśli dane to jedna liczba, to od samych danych,

- dokładność przybliżenia (liczba miejsc dziesiętnych).

Taki aspekt danych, względem którego mierzymy koszt możemy określić za pomocą funkcji μ

Jak połączyć te dwie funkcje? Dla określonego aspektu danych możemy mieć wiele różnych kosztów, zależnie od wyboru konkretnych danych.

$$\vec{\varphi} \circ \mu^{-1} : N \rightarrow \mathcal{P}(N)$$

Wybierając koszt najmniejszy lub największy mówimy o koszcie pesymistycznym lub optymistycznym.

$$\max \circ \vec{\varphi} \circ \mu^{-1} : N \rightarrow N$$

$$\min \circ \vec{\varphi} \circ \mu^{-1} : N \rightarrow N$$

Takie funkcje możemy już porównywać co do rzędów wielkości.

Jak mierzyć koszt średni? D — zmienna losowa. Wówczas $\varphi(D)$ i $\mu(D)$ też są zmiennymi losowymi. Koszt średni, to:

$$E(\varphi(D) \mid \mu(D) = n)$$

Programy niedeterministyczne. Jeśli program jest niedeterministyczny, to dla konkretnych danych może wymagać nie tyle konkretnych ilości zasobów, ale ilości wymaganych zasobów tworzą zbiór. Wówczas φ nie jest funkcją, ale relacją, a powyższe wzory pozostają bez zmian.

9.3 Koszt czasowy

Liczba elementarnych operacji potrzebnych do wyliczenia zadanej wartości, jako funkcja [rozmiaru] danych. Co to są operacje elementarne:

- odczytanie wartości symbolu ze środowiska (stała, liczba) — 1,
- zastosowanie procedury do argumentów — koszt obliczenia wszystkich elementów (w tym procedury) + koszt wyliczenia treści procedury + liczba elementów kombinacji,
- procedury wbudowane — koszt wyliczenia treści = 1, chyba że powiedziane inaczej,
- **if** — koszt obliczenia warunku + zależnie od wyniku koszt obliczenia odpowiedniej części + 1,
- λ -abstrakcja (**function**) — 1,
- dopasowywanie wzorców **match-with** — obliczenie dopasowywanej wartości + łączna długość wzorców + koszt obliczenia wyrażenia odpowiadającego dopasowanemu wzorcowi.
- wyrażenie **let-in** — koszt wyliczenia definiowanych lokalnie wartości + ich liczba + wyliczenie po **in**,
- rekurencja — równanie rekurencyjne na kosztach.

Przykład: Nierekurencyjny przykład z `let`:

```
let f =
  let g x = 3 * x
  in
  function x ->g (2 * x);;
f 7;;
```

Przykład: Koszt czasowy jako funkcja n :

```
let rec silnia n =
  if n < 2 then 1 else n * silnia (n - 1);;
```

9.4 Koszt pamięciowy

Liczymy pamięć zajmowaną przez ramki zawierające potrzebne symbole, pamięć zajmowaną przez tworzone wartości plus pamięć potrzebną do obliczania wyrażeń. Na potrzeby tego wykładu, nie wliczamy danych, natomiast wliczamy wartości pośrednie i wynik. Co to znaczy potrzebne symbole:

- wszystkie symbole widoczne w środowisku (nie przysłonięte) są potrzebne,
- jeśli wartością potrzebnego symbolu jest procedura, to symbole występujące w środowisku wskazywanym przez tę procedurę są potrzebne,
- jeśli potrzebna jest wartość złożona (np. lista lub para), to potrzebne są również jej składowe.

Wartości mogą być współdzielone, tzn. na tę samą wartość może wskazywać kilka wskaźników. Wartości:

- liczby, znaki i wartości logiczne mają rozmiar 1,
- n -tka ma rozmiar n ,
- lista ma rozmiar taki jak jej długość (a w przypadku list nieskończonych — liczba rekordów tworzących listę) + łączne rozmiary elementów + 1,
- wariant — 1 + wielkość ew. argumentu,
- wartość proceduralna — $2 \times$ liczba argumentów (nie liczymy kodu).

Brak ogólnej zasady obliczania kosztu pamięciowego. Należy prześledzić sposób obliczania zgodnie z modelem środowiskowym i określić ile pamięci (maksymalnie) wymaga obliczenie. Należy przy tym uwzględnić współdzielenie wartości i rekurencję ogonową.

Mimo całego skomplikowania liczenia kosztów, zwłaszcza pamięciowego, jest to proste. Działa tu zasada 80%–20%⁴. Ponieważ interesuje nas tylko rząd kosztu, możemy stosować uproszczenia już w trakcie jego wyliczania, oszczędzając sobie masę pracy:

⁴Zasada 80%–20% mówi, że 80% rezultatu można osiągnąć nakładem 20% pracy, a pozostałe 20% rezultatu wymaga 80% nakładów pracy. Zasada ta ma zastosowanie do bardzo wielu dziedzin, w tym do programowania.

- koszt stały = $\Theta(1)$,
- w równaniach rekurencyjnych, opisujących koszty funkcji rekurencyjnych, składniki nie zawierające odwołań rekurencyjnych możemy zastąpić równymi co do rzędu,
- czasami prościej jest oszacować rząd funkcji niż ją dokładnie wyliczyć,
- koszt pamięciowy nie może być większy niż koszt czasowy.

9.5 Przykład: Potęgowanie

Możemy skorzystać ze wzorów:

$$\begin{aligned} b^n &= b \cdot b^{n-1} \\ b^0 &= 1 \end{aligned}$$

Co możemy zaimplementować od razu jako:

```
let rec potega b n =
  if n = 0 then 1 else b * potega b (n-1);;
```

Liczba kroków jest równa $n + 1$, każdy krok ma stały koszt i wymaga stałej ilości pamięci. Koszt czasowy i pamięciowy są rzędu $T(n) = M(n) = \Theta(n)$.

Koszt pamięciowy możemy polepszyć do $M(n) = \Theta(1)$ stosując rekurencję ogonową:

```
let potega b n =
  let rec iter n a =
    if n = 0 then a else iter (n-1) (a*b)
  in
  iter n 1;;
```

przy czym $\text{iter } n \ a = a \cdot b^n$. Koszt czasowy pozostaje jednak bez zmian.

Możemy jednak skorzystać z innego wzoru na potencowanie:

$$\begin{aligned} b^0 &= 1 \\ b^{2n} &= (b^2)^n \\ b^{2n+1} &= b \cdot b^{2n} \end{aligned}$$

który zapisujemy jako:

```
let potega b n =
  let rec pot b n a =
    if n = 0 then a
    else if parzyste n then pot (square b) (n / 2) a
    else pot (square b) ((n - 1)/2) (a * b)
  in
  pot b n 1;;
```

przy czym $\text{pot } b \ n \ a = a \cdot b^n$. Mamy tutaj rekurencję ogonową, więc koszt pamięciowy jest stały, $M(n) = \Theta(1)$. Jaka jest jednak liczba kroków? Można pokazać przez indukcję, że jeżeli $2^k \leq n < 2^{k+1}$, to algorytm wymaga $k + 2$ kroków. Stąd, $T(n) = \Theta(\log n)$.

9.6 Przykład: Algorytm mnożenia rosyjskich chłopów

Oto przykład rekurencyjnego algorytmu mnożenia, który był używany przez chłopów w pewnych obszarach Syberii do mnożenia w pamięci. Algorytm opiera się na dwóch tożsamościach:

$$x \cdot y = \frac{x}{2} \cdot 2y \quad \text{dla } x \text{ parzystych}$$
$$x \cdot y = (x - 1) \cdot y + y \quad \text{dla } x \text{ nieparzystych}$$

```
let razy x y =
  let rec pom x y a =
    if x = 0 then
      a
    else if parzyste x then
      pom (x / 2) (2 * y) a
    else
      pom (x - 1) y (a + y)
  in
  if x > 0 then
    pom x y 0
  else
    pom (-x) (-y) 0;;
```

Dzięki rekurencji ogonowej w procedurze `pom` złożoność pamięciowa tego algorytmu jest rzędu $\Theta(1)$. (Inaczej nie dałoby się go stosować do mnożenia w pamięci.) Pokażemy, że złożoność czasowa jest rzędu $\Theta(\max(\log |x|, 1))$. Wystarczy, że skupimy się na procedurze `pom`, gdyż narzut wprowadzony przez procedurę `razy` jest stały. Każdy krok tej procedury działa w stałym czasie. Pozostaje pytanie, ile jest tych kroków?

Jeśli x jest nieparzyste, to w kolejnym kroku jest parzyste. Przynajmniej w co drugim kroku x jest parzyste. Jeśli zaś jest parzyste, to w kolejnym kroku x maleje o połowę (z wyjątkiem ostatniego kroku, gdy $x = 0$). Tak więc kroków, gdy x jest parzyste jest co najwyżej $\log x + 1$, czyli złożoność czasowa jest rzędu $O(\max(\log |x|, 1))$.

Z drugiej strony, dla x nieparzystego, jego wartość maleje tylko o 1, a więc liczba wszystkich kroków nie może być mniejsza niż $\log x + 1$, czyli złożoność czasowa jest rzędu $\Omega(\max(\log |x|, 1))$. Reasumując, złożoność czasowa jest rzędu $\Theta(\max(\log |x|, 1))$.

9.7 Przykład: Brakująca wartość na liście liczb [Bentley]

Rozważmy następujący problem: Mamy daną dodatnią liczbę całkowitą n oraz listę mniej niż n liczb całkowitych z zakresu od 1 do n . Z zasady szufladkowej Dirichleta wynika, że przynajmniej jednej liczby całkowitej od 1 do n musi na tej liście brakować. Należy znaleźć jedną z takich brakujących wartości.

Możemy tu zastosować technikę bisekcji: Podzielmy przedział od 1 do n na dwa przedziały (prawie) równej wielkości. W jednym z tych przedziałów musi brakować jakiejś wartości. Podzielmy listę na dwie, w zależności od tego, do którego przedziału należą elementy. Porównując długość list, możemy zawęzić poszukiwania do krótszego przedziału i krótszej listy.

Jak to zwykle bywa w przypadku bisekcji, trzeba bardzo uważać na warunki brzegowe. Jeżeli dzielony przedział zawiera nieparzystą liczbę elementów, a listy uzyskane w wyniku podziału mają równe długości, to należy wybrać większą „połówkę” przedziału.

```

let szukaj l n =
  let rec binary l a b =
    if l = [] then a
    else
      let c = (a + b) / 2
      in
        let l1 = filter (fun x ->x <= c) l
        and l2 = filter (fun x ->x >c) l
        in
          if length l1 <= length l2 then
            binary l1 a c
          else
            binary l2 (c+1) b
  in
    binary l 1 n;;

```

Oszacowanie złożoności pamięciowej nie jest trudne. Dzięki rekurencji ogonowej, złożoność pamięciowa jest taka, jak rozmiar przekazywanej listy, czyli $M(n) = \Theta(n)$.

Oszacowanie złożoności czasowej jest trochę bardziej skomplikowane. Nie zmieniając rzędu złożoności, możemy ją oszacować nierównością rekurencyjną:

$$\begin{aligned}
 T(1) &= 1 \\
 T(n) &\leq n + T\left(\left\lceil \frac{n}{2} \right\rceil\right)
 \end{aligned}$$

Stąd $T(n) \leq n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots = O(n)$. Z drugiej strony, koszt pierwszego wywołania procedury `binary` jest rzędu $\Theta(n)$. Stąd $T(n) = \Theta(n)$.

9.8 Przykład: Algorytm Euklidesa

9.8.1 Algorytm Euklidesa przez odejmowanie

```

let rec nwd x y =
  if x = y then x else
  if x > y then
    nwd (x - y) y
  else
    nwd x (y - x);;

```

Dla $x, y > 0$ mamy $(\text{nwd } x \ y) = \text{NWD}(x, y)$.

Jaka jest złożoność czasowa (ze względu na $n = x + y$)? Liczba wykonywanych kroków może być liniowa, np. dla $(\text{nwd } x \ 1)$, czyli $T(n) = \Omega(n)$. Gorsza nie będzie, bo z każdym krokiem maleje wartość $x + y$, $T(n) = O(n)$, czyli $T(n) = \Theta(n)$. Mamy tu do czynienia z rekurencją ogonową, więc złożoność pamięciowa jest stała, $M(n) = \Theta(1)$.

9.8.2 Algorytm Euklidesa przez dzielenie

```

let nwd a b =
  let rec e a b =

```

```

    if b = 0 then a else e b (a mod b)
in
    if a >b then e a b else e b a;;

```

Jakie są wymagania wobec argumentów? ($\max(a, b) > 0$, $\min(a, b) \geq 0$) Ile operacji arytmetycznych wykona ten algorytm (ze względu na $\min(a, b)$)?

Lemat 1 (Lame). *Oznaczmy przez (a_i, b_i) pary wartości a i b , dla których powyższy algorytm wykonuje i kroków. Wówczas $b_i \geq \text{Fib}_{i-1}$.*

Dowód. Dowód indukcyjny.

1. jeśli jeden krok, to $b_1 = 0$,
2. jeśli dwa kroki, to $b \geq 1$,
3. jeśli więcej kroków, to $(a_{k+1}, b_{k+1}) \rightarrow (a_k, b_k) \rightarrow (a_{k-1}, b_{k-1})$, to $a_k = b_{k+1}$, $a_{k-1} = b_k$, $b_{k-1} = a_k \pmod{b_k}$, czyli $a_k = qb_k + b_{k-1}$ dla $q \geq 1$, czyli $b_{k+1} \geq b_k + b_{k-1}$.

□

Liczby Fibonacciego można przybliżać wzorem:

$$\text{Fib}_n \approx \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

czyli koszt czasowy $T(a + b) = O(\log(a + b))$. Z drugiej strony mamy

$$\text{Fib}_{n+1} \pmod{\text{Fib}_n} = (\text{Fib}_n + \text{Fib}_{n-1}) \pmod{\text{Fib}_n} = \text{Fib}_{n-1}$$

czyli dla $a = \text{Fib}_{n+1}$ i $b = \text{Fib}_n$ algorytm wykonuje n kroków. Stąd $T(a + b) = \Theta(\log(a + b))$.

Koszt pamięciowy ze względu na rekurencję ogonową wynosi $M(n) = \Theta(1)$.

9.9 Algorytm Euklidesa przez parzystość

Zastosowanie zasady dziel i rządź.

```

let nwd x y =
  let rec pom x y a =
    if x = y then a * x
    else if parzyste x && parzyste y then pom (x / 2) (y / 2) (2 * a)
    else if parzyste x then pom (x / 2) y a
    else if parzyste y then pom x (y / 2) a
    else if x >y then pom (x - y) y a else pom x (y - x) a
  in
    pom x y 1;;

```

Algorytm ten wykorzystuje jedynie odejmowanie i mnożenie/dzielenie/modulo 2. Ze względu na zapis binarny liczb operacje te mają koszt $O(\text{długość zapisu liczby})$, nawet dla bardzo dużych liczb. Jakie są wymagania wobec argumentów? ($a, b > 0$) Ile operacji arytmetycznych wykona ten algorytm?

Oznaczmy przez (a_i, b_i) pary wartości a i b , dla których powyższy algorytm wykonuje i kroków. Mamy $a_{i+1} \geq a_i$, $b_{i+1} \geq b_i$, $b_1 = a_1 > 0$. W pierwszych trzech przypadkach $a_{i+1}b_{i+1} \geq 2a_ib_i$. W czwartym przypadku $a_{i+1}b_{i+1} \geq 2a_{i-1}b_{i-1}$. Przez indukcję pokazujemy, że

$$a_ib_i \geq 2^{\lfloor \frac{i-1}{2} \rfloor}$$

Stąd algorytm wykona $O(\log(ab)) = O(\log \max(a, b))$ kroków.

Z drugiej strony, $a_{i+1} + b_{i+1} \leq 2(a_i + b_i)$. Czyli $a_i + b_i \leq 2^{i-1}(a_1 + b_1)$. Stąd algorytm wykona $\Omega(\log(a+b)) = \Omega(\log \max(a, b))$ kroków. Tak więc algorytm wykona $\Theta(\log \max(a, b))$ kroków.

Każdy krok niesie ze sobą stały koszt czasowy, stąd koszt czasowy wynosi $T(\max(a, b)) = \Theta(\log \max(a, b))$. Mamy tu do czynienia z rekurencją ogonową, stąd stała złożoność pamięciowa, $M(\max(a, b)) = \Theta(1)$.

Jakie będą koszty algorytmu w przypadku długich liczb? (*długość liczb)

9.10 Przykład: Liczby Fibonacciego

Przyjrzyjmy się różnym algorytmom liczenia liczb Fibonacciego. Najprostszy z nich, i zarazem najmniej efektywny, opiera się na rekurencyjnym wzorze definiującym liczby Fibonacciego.

$$Fib_0 = 0 \quad Fib_1 = 1 \quad Fib_{n+1} = Fib_n + Fib_{n-1}$$

```
let rec fib n =
  if n < 2 then n else fib (n - 1) + fib (n - 2);;
```

Jaka jest złożoność czasowa tego rozwiązania? Wyobraźmy sobie, że obliczając Fib_n rozwijamy podaną definicję rekurencyjną, aż do uzyskania sumy zer i jedynek. Uzyskamy sumę zawierającą Fib_n jedynek i nie więcej niż Fib_n zer. Wiemy (z *Matematyki dyskretnej*), że $Fib_n \approx \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}}$. Tak więc złożoność czasowa tego algorytmu to $T(n) = \Theta(Fib_n) = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$. Ponieważ rekurencja w rozwiązaniu nie jest ogonowa, więc złożoność pamięciowa jest takiego rzędu, jak głębokość rekurencji, czyli $M(n) = \Theta(n)$.

Bardziej efektywny algorytm pamięta dwie kolejne liczby Fibonacciego i wykorzystuje rekurencję ogonową:

```
let fib n =
  let rec fib_pom a b n =
    if n = 0 then
      a
    else
      fib_pom b (a + b) (n - 1)
  in
  fib_pom 0 1 n;;
```

Ze względu na rekurencję ogonową złożoność pamięciowa jest stała, $M(n) = \Theta(1)$. Iteracja wykonuje $n + 1$ kroków, każdy w stałym czasie, stąd $T(n) = \Theta(n)$.

Jeszcze szybsze rozwiązanie uzyskamy wykorzystując mnożenie macierzy i następujące tożsamości:

$$F = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \quad F \times \begin{pmatrix} Fib_i \\ Fib_{i+1} \end{pmatrix} = \begin{pmatrix} Fib_{i+1} \\ Fib_{i+2} \end{pmatrix}$$

$$F^n \times \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} Fib_n \\ Fib_{n+1} \end{pmatrix} \quad F^n = \begin{pmatrix} Fib_{n-1} & Fib_n \\ Fib_n & Fib_{n+1} \end{pmatrix}$$

Problem sprowadza się do obliczenia macierzy F^n , co możemy zrobić analogicznie do potęgowania liczb. Ponieważ macierze 2×2 mnożyć w czasie stałym, uzyskujemy złożoność czasową $T(n) = \Theta(\log n)$. Dzięki rekurencji ogonowej, złożoność pamięciowa jest stała, $M(n) = \Theta(1)$.

```
let mnoz ((x11,x12),(x21,x22)) ((y11,y12),(y21,y22)) =
  ((x11 * y11 + x12 * y21,x11 * y12 + x12 * y22),
   (x21 * y11 + x22 * y21,x21 * y12 + x22 * y22));;
```

```
let square x = mnoz x x;;
```

```
let f = ((0,1),(1,1));;
```

```
let id = ((1,0),(0,1));;
```

```
let potega b n =
  let rec pot b n a =
    if n = 0 then a
    else if parzyste n then pot (square b) (n / 2) a
    else pot b (n - 1) (mnoz a b)
  in
  pot b n id;;
```

```
let fib n =
  let ((_,v),_) = potega f n
  in v;;
```

9.11 Przykład: logarytm całkowitoliczbowy

Rozważmy następujący problem: Dana jest dodatnia liczba całkowita n . Należy obliczyć logarytm całkowitoliczbowy przy podstawie 2 z n , czyli $\lfloor \log_2 n \rfloor$. Najprostsze rekurencyjne rozwiązanie tego zadania ma postać:

```
let rec int_log n =
  if n = 1 then 0
  else 1 + int_log (n / 2);;
```

Z każdym krokiem iteracji n jest dzielone przez 2. Kroków tych nie będzie więcej niż $\log_2 n + 1$, przy czym dla $n = 2^k$ jest ich dokładnie tyle. Czyli złożoność czasowa jest rzędu $T(n) = \Theta(\log n)$. Ponieważ rekurencja nie jest ogonowa, złożoność pamięciowa jest taka sama, jak czasowa $M(n) = \Theta(\log n)$.

Złożoność pamięciową łatwo poprawić do $M(n) = \Theta(1)$, stosując rekurencję ogonową:

```

let int_log n =
  let rec pom n a =
    if n = 1 then a
    else pom (n / 2) (a + 1)
  in pom n 0;;

```

To rozwiązanie można dalej polepszyć. Pomyślmy o zapisie binarnym liczby $\lfloor \log_2 n \rfloor$. Powiedzmy, że najstarszy bit tej liczby znajduje się na pozycji i . Liczbę i możemy wyznaczyć porównując n z liczbami postaci $2^1, 2^2, 2^4, \dots, 2^{2^i}, 2^{2^{i+1}}$. Znając liczby $i, 2^i$ oraz 2^{2^i} korzystamy z tożsamości:

$$\lfloor \log_2 n \rfloor = 2^i + \lfloor \log_2 \frac{n}{2^{2^i}} \rfloor$$

i sprowadzamy problem do obliczenia $\lfloor \log_2 \frac{n}{2^{2^i}} \rfloor$.

```

let int_log n =
  let rec pom i j k m acc =
    if m = 1 then acc
    else if square k > m then
      pom 0 1 2 (m / k) (acc + j)
    else
      pom (i+1) (2*j) (square k) m acc
  in pom 0 1 2 n 0;;

```

Dla procedury pomocniczej `pom` spełniony jest następujący niezmiennik: $j = 2^i, k = 2^j, k \leq m$, oraz następujący warunek końcowy: `pom i j k m acc = acc + $\lfloor \log_2 m \rfloor$` .

Wyznaczenie pozycji i najstarszego bitu wyniku wymaga $i + 1$ kroków. Wyznaczając kolejny bit wyniku nie korzystamy z wcześniej obliczonych wartości, tylko wyznaczamy go od początku, jako najstarszy bit liczby $\frac{n}{2^{2^i}}$, itd. Stąd złożoność czasowa algorytmu jest rzędu:

$$T(n) = O\left(\sum_{i=1}^{\log \log n + 1} i\right) = O((\log \log n)^2)$$

Z drugiej strony, dla $n = 2^{2^i - 1}$ taki czas działania jest osiągalny, czyli $T(n) = \Theta((\log \log n)^2)$. Dzięki zastosowaniu rekurencji ogonowej złożoność pamięciowa jest rzędu $M(n) = \Theta(1)$.

Złożoność czasową można jeszcze polepszyć. Wyznaczając pozycję i najstarszego bitu wyniku obliczamy liczby $2^1, 2^2, 2^4, \dots, 2^{2^i}, 2^{2^{i+1}}$. Jeśli je zapamiętamy, to możemy je wykorzystać do wyznaczenia kolejnych bitów wyniku. Podzielmy algorytm na dwie fazy. W pierwszej (procedura `gen`) generujemy ciąg (listę) par liczb postaci $[(2^i, 2^{2^i}); \dots (4, 2^4); (2, 2^2); (1, 2^1)]$. W drugiej fazie (procedura `scan`), korzystając z takiego ciągu par, wyznaczamy wynik, w kolejności od jego najstarszego bitu do najmłodszego.

```

let int_log n =
  let rec gen i j k acc =
    if k > n then acc
    else gen (i+1) (2*j) (square k) ((j,k) :: acc)
  and scan l m =
    match l with
    [] -> 0 |

```

```

(j,k)::t ->
  if k <= m then
    j + scan t (m / k)
  else
    scan t m
in scan (gen 0 1 2 []) n;;

```

Dla procedury pomocniczej **gen** spełniony jest następujący niezmiennik: $j = 2^i$, $k = 2^j$, $\sqrt{k} \leq n$, $acc = [(2^{i-1}, 2^{2^{i-1}}); \dots (2, 2^2); (1, 2^1)]$, oraz warunek końcowy: **gen** $i j k acc = acc + \lfloor \log_2 m \rfloor$. Dla procedury pomocniczej **scan** spełniony jest następujący niezmiennik: $l = [(2^i, 2^{2^i}); \dots (2, 2^2); (1, 2^1)]$, $m < 2^{2^{i+1}}$, oraz następujący warunek końcowy: **scan** $l m = \lfloor \log_2 m \rfloor$. Złożoność czasowa (obu procedur pomocniczych, oraz całego algorytmu) jest rzędu $T(n) = \Theta(\log \log |n|)$. Ze względu na rozmiar konstruowanej listy oraz brak rekurencji ogonowej w procedurze **scan**, złożoność pamięciowa jest również rzędu $M(n) = \Theta(\log \log |n|)$.

Mamy tu do czynienia z polepszeniem złożoności czasowej kosztem pogorszenia złożoności pamięciowej (ang. *time-memory trade-off*). Zwykle, bardziej zależy nam na polepszeniu złożoności czasowej, niż pamięciowej. Tak więc to rozwiązanie jest lepsze od poprzedniego.

A czy możemy je jeszcze polepszyć? Tak, możemy zbić złożoność pamięciową do $M(n) = \Theta(1)$. Problem polega na tym, że o ile łatwo możemy przejść od liczb $i, 2^i, 2^{2^i}$ do liczb $i+1, 2^{i+1}, 2^{2^{i+1}}$, o tyle przejście w drugą stronę nie jest już takie proste. Rozwiązanie polega na tym, aby rozpatrywać liczby postaci $i, Fib_i, Fib_{i+1}, 2^{Fib_i}, 2^{Fib_{i+1}}$. Podobnie jak poprzednio, mamy dwie fazy. W pierwszej (procedura **gen**) generujemy taki zestaw liczb $i, Fib_i, Fib_{i+1}, 2^{Fib_i}, 2^{Fib_{i+1}}$, że $2^{Fib_i} \leq n < 2^{Fib_{i+1}}$. W drugiej fazie (procedura **scan**), korzystając z takiego zestawu liczb, wyznaczamy wynik. Jeżeli pomyślimy o zapisie wyniku w systemie Zeckendorfa⁵, to druga faza wyznacza kolejne cyfry wyniku w kolejności od najbardziej, do najmniej znaczącej.

```

let int_log n =
  let rec gen i j k l m =
    (* j = Fib_i, k = Fib_(i+1), l = 2^Fib_i, m = 2^Fib_(i+1), l <= n, *)
    if m > n then
      scan i j k l m n 0
    else
      gen (i+1) k (j+k) m (l*m)
  and scan i j k l m nn acc =
    (* j = Fib_i, k = Fib_(i+1), l = 2^Fib_i, m = 2^Fib_(i+1), m > nn, int-log n = acc + int-log nn *)
    if nn = 1 then acc
    else if l <= nn then
      scan (i-1) (k-j) j (m/l) l (nn/l) (acc+j)
    else
      scan (i-1) (k-j) j (m/l) l nn acc
  in
  gen 1 1 1 2 2;;

```

Dla procedury pomocniczej **gen** spełniony jest następujący niezmiennik: $j = Fib_i$, $k = Fib_{i+1}$, $l = 2^{Fib_i}$, $m = 2^{Fib_{i+1}}$, $l \leq n$, oraz warunek końcowy: **gen** $i j k l m = \lfloor \log_2 n \rfloor$. Dla

⁵System binarny, w którym i -ty bit ma wagę Fib_{i+1} . W systemie tym żadne dwie jedyńki nie sąsiadują ze sobą.

procedury pomocniczej `scan` spełniony jest następujący niezmiennik: $j = \text{Fib}_i$, $k = \text{Fib}_{i+1}$, $l = 2^{\text{Fib}_i}$, $m = 2^{\text{Fib}_{i+1}}$, $m > nn$, $\lfloor \log_2 n \rfloor = \text{acc} + \lfloor \log_2 nn \rfloor$, oraz następujący warunek końcowy: `scan i j k l m nn acc = $\lfloor \log_2 nn \rfloor + \text{acc}$` . Ponieważ ciąg liczb Fibonacciego rośnie wykładniczo szybko, złożoność czasowa (obu procedur pomocniczych, oraz całego algorytmu) jest rzędu $T(n) = \Theta(\log \log n)$. Dzięki zastosowaniu rekurencji ogonowej złożoność pamięciowa jest rzędu $M(n) = \Theta(1)$.

9.12 Test na liczby pierwsze

Jak sprawdzić czy dana liczba jest liczbą pierwszą? Należy sprawdzić, czy ma jakieś dzielniki, do \sqrt{n} .

```
let min_dzielnik n =
  let rec dziel k =
    if square k > n then n
    else if (n mod k) = 0 then k
    else dziel (k + 1)
  in
  dziel 2;;
```

Dzięki rekurencji ogonowej koszt pamięciowy jest stały. Liczba kroków nie przekracza \sqrt{n} , czyli złożoność czasowa jest rzędu $O(\sqrt{n})$.

9.13 Test Fermata i Millera-Rabina

Oto algorytm Millera-Rabina sprawdzania, czy dana liczba jest pierwsza.

Tw. 1 (Małe twierdzenie Fermata). *Jeśli n jest liczbą pierwszą, a a jest dowolną dodatnią liczbą mniejszą niż n , to $a^{n-1} \equiv 1 \pmod n$.*

Wniosek: jeśli $1 < a < n - 1$ i $a^{n-1} \not\equiv 1 \pmod n$, to n nie jest liczbą pierwszą. Niestety istnieją takie liczby n , które nie są pierwsze, ale dla każdego $1 < a < n - 1$ zachodzi $a^{n-1} \equiv 1 \pmod n$. Liczby takie nazywamy liczbami *Carmichaela*. Oto kilka pierwszych liczb Carmichaela: 561, 1105, 1729.

Fakt 1. *Jeśli n ma nietrywialny pierwiastek 1, czyli istnieje takie $1 < k < n - 1$, że $k^2 \equiv 1 \pmod n$, to n nie jest liczbą pierwszą.*

Dowód. Niech p będzie tym nietrywialnym pierwiastkiem. Wówczas $(p-1) \cdot (p+1) = p^2 - 1 = 0$. Równocześnie $p-1 \neq 0$ i $p+1 \neq 0$, a więc Z_n nie jest ciałem, czyli n nie jest liczbą pierwszą. \square

Przykład: Weźmy $n = 15$. Ewidentnie $15 = 3 \cdot 5$ nie jest liczbą pierwszą. Mamy następujące pierwiastki jedynki: $1^2 = 1 \equiv 1 \pmod{15}$, $4^2 = 16 \equiv 1 \pmod{15}$, $11^2 = 121 \equiv 1 \pmod{15}$, $14^2 = 196 \equiv 1 \pmod{15}$. Przy tym 4 i 11 nie są trywialnymi pierwiastkami z 1.

Nasz algorytm polega na wylosowaniu liczby a i sprawdzeniu, czy spełnione jest tw. Fermata. Ponadto obliczając $a^{n-1} \pmod n$ szukamy nietrywialnych pierwiastków z 1. Jeżeli oba kryteria są spełnione, przyjmujemy, że n jest pierwsze. Możemy się pomylić, fałszywie przyjmując, że n jest pierwsze.

Obliczając $a^{n-1} \pmod n$ sprawdzamy wszystkie pojawiające się wartości, czy nie są nietrywialnymi pierwiastkami z 1. Przypadki, gdy natrafiamy na taki pierwiastek sygnalizujemy podnosząc wyjątek.

```
exception Pierwiastek;;

let rec expmod b k n =
  let test x =
    if not (x = 1) && not (x = n - 1) && (square x) mod n = 1 then
      raise Pierwiastek
    else
      x
  in
  if k = 0 then 1 else
    if parzyste k then (square (test (expmod b (k / 2) n))) mod n
    else ((test (expmod b (k-1) n)) * b) mod n;;
```

Można pokazać, że jeżeli n jest liczbą nieparzystą i nie jest liczbą pierwszą, to przynajmniej dla połowy $1 < a < n - 1$ obliczenie $a^{n-1} \pmod n$ odkryje nietrywialny pierwiastek 1. Tak poprawiony test Fermata jest znany jako test Millera-Rabina.

```
let randtest n =
  if parzyste n then
    n = 2
  else if n = 3 then true
  else
    try
      expmod (Random.int (n-3) + 2) (n-1) n = 1
    with Pierwiastek ->false;;
```

Z pewnym prawdopodobieństwem test ten może stwierdzić, że liczba, która nie jest pierwsza, jest pierwsza. Spróbujmy oszacować prawdopodobieństwo pomyłki. Załóżmy, że n nie jest liczbą pierwszą i $n > 3$. Dodatkowo założmy, że dla wylosowanego a spełnione jest kryterium Fermata. Prawdopodobieństwo pomyłki możemy oszacować z góry prawdopodobieństwem pomyłki testu na nietrywialne pierwiastki z 1. To prawdopodobieństwo nie przekracza natomiast $\frac{1}{2}$. Tak więc test ten z prawdopodobieństwem co najmniej $\frac{1}{2}$ daje poprawną odpowiedź (a tak naprawdę z dużo większym), a z prawdopodobieństwem nieprzekraczającym $\frac{1}{2}$ uzna liczbę, która nie jest pierwsza, za pierwszą.

Powtarzając taki test k zmniejszamy prawdopodobieństwo błędu poniżej $(\frac{1}{2})^k$. Jeśli chcemy być pewni wyniku z prawdopodobieństwem p , to powtarzamy ten test $-\log_2(1-p)$ razy. Koszt pojedynczego testu, pamięciowy i czasowy, jest rzędu $\Theta(\log n)$. Ponieważ liczba powtórzeń testu jest stała, więc koszt całego algorytmu jest rzędu $\Theta(\log n)$.

[[Rozwinąć: gotowe zestawy wartości a , które wystarczają dla ograniczonych int-ów.]]

9.14 Algorytmy Monte Carlo i Las Vegas

Rozróżniamy dwie klasy algorytmów randomizowanych:

- Monte Carlo — złożoność zawsze dobra, ale z małym prawdopodobieństwem może dawać złe wyniki, lub wyniki nieznacznie zaburzone,

- Las Vegas — zawsze daje dobre wyniki, ale z małym prawdopodobieństwem działa dłużej; średnia złożoność musi być OK.

Ćwiczenia

1. Piramida to ostrosłup, którego podstawa jest kwadratem, a boczne ściany to trójkąty równoboczne. Zlecono Ci pomalowanie bocznych ścian piramidy. Malując piramidę, możesz wziąć ze sobą wiaderko farby, które starcza na pomalowanie $1 m^2$ powierzchni, co trwa 1 minutę. Zarówno wejście na wysokość h metrów, jak i zejście na dół, trwają po $\frac{h}{2}$ minut. Podstawa piramidy ma długość n metrów. Podaj, jakiego rzędu jest czas potrzebny do pomalowania całej piramidy.
2. Jaka jest asymptotycznie energia potencjalna (wypełnionej) piramidy?
3. Jedziesz w ciemności samochodem. W odległości n metrów przed samochodem idzie pieszy. Jak ilość światła, która dociera do oczu kierowcy zależy od n ?
 - Jeżeli pieszy nie ma odbłasków, nie jest ciałem doskonale czarnym i równomiernie rozprasza padające na niego światło.
 - Jeżeli pieszy ma (idealny) odbłask, który padające na niego światło odbija dokładnie tam skąd ono pada (a kierowca ma reflektory w oczach).
4. Zastanówmy się, jaka energia jest potrzebna do napompowania koła rowerowego. Dla ustalonej objętości dętki, jakiego rzędu jest ta energia w zależności od ciśnienia?
5. Ciśnienie powietrza jest takie, jak ciężar słupa powietrza naciskającego z góry. Chcemy się wznieść na taką wysokość, na której ciśnienie powietrza nie przekracza p . Jakiego rzędu jest to wysokość?
6. Dana jest para funkcji f i g , z liczb całkowitych w liczby całkowite. Wiadomo, że f jest ściśle rosnąca, a g jest ściśle malejąca. Napisz taką procedurę `znajdz` : $(\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int}) \rightarrow \text{int}$, że dla $w = \text{znajdz } f \ g$, mamy:

$$|f(w) - g(w)| = \min_{i \in \mathbb{Z}} |f(i) - g(i)|$$

Możesz założyć, że istnieją takie liczby całkowite i i i' , że $f(i) < g(i)$ oraz $f(i') > g(i')$. Podaj złożoność czasową i pamięciową swojego rozwiązania.

7. Dana jest implementacja funkcji `sprawdz` o treści:

```
let sprawdz (a:int) =  
  
    let n = ???  
  
    in if a < n then -1 else if a = n then 0 else 1;;
```

W tej implementacji pod `???` została ukryta pewna liczba całkowita. Napisz funkcję `znajdz`, która odwołując się do funkcji `sprawdz`, znajdzie tę liczbę całkowitą. Podaj złożoność czasową i pamięciową rozwiązania.

Uwaga: Zakładamy, że typ `int` reprezentuje dowolne liczby całkowite. W szczególności nie można zakładać, że typ `int` jest skończony i tym samym używać stałych takich jak `max_int`.

8. [CEOI 2003, uproszczone zadanie Hanoi] Wiadomo (z EMD), że żeby przenieść n krążków w łamigłówce „wieże Hanoi” trzeba wykonać $2^n - 1$ ruchów. Napisz procedurę `hanoi: int list -> int -> int`, która dla zadanej konfiguracji krążków oraz słupka, na którym początkowo znajdują się wszystkie krążki wyznaczy minimalną liczbę ruchów potrzebnych do uzyskania danej konfiguracji.

Słupki są reprezentowane jako liczby całkowite od 1 do 3. Konfiguracja to lista numerów słupków, na których mają się znaleźć krążki, w kolejności od największych krążków do najmniejszych.

9. [VIII OI, zadanie *Łańcuch*] (Przynieść i pokazać chiński łańcuch.) W jednym ruchu można zawsze zdjąć lub założyć pierwsze ogniwo łańcucha. Ponadto, jeżeli zdjęte są ogniwa o numerach $1, 2, \dots, k-2$, ogniwo nr $k-1$ jest założone, to w jednym ruchu można zdjąć lub założyć ogniwo nr k . Początkowo wszystkie ogniwa łańcucha są założone. Napisz procedurę, której wynikiem jest lista ruchów prowadzących do zdjęcia n ogniw łańcucha. Oblicz złożoność czasową i pamięciową rozwiązania. Jeśli złożoność czasowa jest większa od pamięciowej ($\Theta(2^n)$), to popraw program używając akumulatora i nie używając sklejania list.

Wytyczne dla prowadzących ćwiczenia

- Ad. 1** Oczywiście jest to $\Theta(n^3)$. Wyobraźmy sobie pomalowany fragment ściany piramidy. Jego pomalowanie zajmuje czas proporcjonalny do pionowego wycinka piramidy wyznaczonego przez ten fragment ściany. Tak więc pomalowanie całej piramidy zajmuje czas proporcjonalny do jej objętości.
- Ad. 9** Jeśli złożoność czasowa jest większa od pamięciowej ($\Theta(2^n)$), to należy poprawić program używając akumulatora i nie używając sklejania list.

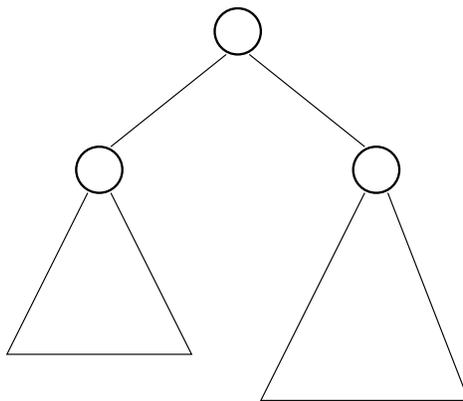
Wykład 10. Zasada „dziel i rządź” i analiza złożoności na przykładzie algorytmów sortowania

Sformułowanie problemu. Dany jest ciąg n elementów, ze zbioru (nieograniczonej mocy), na którym jest określony porządek liniowy \leq i (oprócz definiowania wartości) jest to jedyna dostępna operacja na elementach. Należy utworzyć uporządkowany ciąg elementów, będący permutacją danego ciągu. Koszt porównania elementów jest stały. Chcemy oszacować koszt pesymistyczny sortowania.

Eliminacja algorytmów randomizowanych Jeśli nasz algorytm jest randomizowany, to ustalamy z góry ciąg losowanych liczb i analizujemy go, jak deterministyczny.

Drzewa decyzyjne Ustalmy n . Drzewo decyzyjne to drzewo binarne. W węzłach wewnętrznych mamy operacje porównania elementów na konkretnych pozycjach. W liściach są permutacje określające wyniki sortowania. Obliczeniu odpowiada przejście ścieżki od korzenia drzewa do liścia.

Fakt 2. *Jeśli drzewo binarne ma k liści, to jego wysokość jest nie mniejsza niż $\log_2 k$.*



Rysunek 1: Przynajmniej jedno z poddrzew drzewa binarnego zawiera przynajmniej połowę liści drzewa.

Dowód. Dowód przebiega indukcyjnie po liczbie liści.

1. $k = 1$: Drzewo ma tylko jeden węzeł — liść.
2. $k > 0$: Jedno z poddrzew ma przynajmniej $\lceil \frac{k}{2} \rceil$ liści (rys. 1). Z założenia indukcyjnego, jego wysokość jest nie mniejsza niż $\log_2 \frac{k}{2}$. Stąd całe drzewo ma wysokość przynajmniej $\log_2 k$.

□

Lemat 2. $3^n \cdot n! \geq n^n$

Dowód. Dowód przebiega indukcyjnie.

1. Dla $n = 1$ mamy $3 \geq 1$.
2. Dla $n > 1$ mamy:

$$\begin{aligned}
3^{n+1}n!(n+1) &\geq 3(n+1)n^n \geq \\
&\geq e(n+1)n^n \geq \\
&\geq \left(1 + \frac{1}{n}\right)^n (n+1)n^n = \\
&= \frac{(n+1)^n}{n^n} (n+1)n^n = (n+1)^{n+1}
\end{aligned}$$

□

Tw. 2. Pesymistyczny koszt sortowania jest rzędu $\Omega(n \log n)$.

Dowód. Rozpatrujemy wszystkie możliwe wykonania naszego algorytmu dla ciągu n różnych elementów. Przebieg obliczenia zależy tylko od porównań tych elementów. Wszystkie takie obliczenia możemy przedstawić sobie w postaci drzewa decyzyjnego, w którego węzłach mamy porównania danych elementów.

Drzewo to ma co najmniej $n!$ liści, gdyż mamy $n!$ możliwych wyników. Z lematu i faktu wynika, że wysokość drzewa h

$$h \geq \log_2 n! \geq \log_2 \left(\frac{n^n}{3^n}\right) = n \log_2 \left(\frac{n}{3}\right) = \Omega(n \log n)$$

□

Analiza kilku algorytmów sortowania. Jak realizujemy zasadę dziel i rządź.

10.1 Selection sort

Podział: wybór maksimum + posortowanie krótszego ciągu + sklejenie.

```

let select_max (h::t) =
  fold_left
    (fun (m,r) x ->if x >m then (x,m::r) else (m,x::r))
    (h, []) t;;

let selection_sort l =
  let przenies (s,l) =
    let (m,r) = select_max l
    in (m::s,r)
  in
  iteruj
    ([],l)
    przenies
    (fun (_,l) ->l = [])
    (fun (s,_) ->s);;

```

Maksimum w liście n -elementowej możemy znaleźć w czasie $\Theta(n)$. Stąd złożoność czasowa tego algorytmu wynosi:

$$\begin{aligned} T(0) &= \Theta(1) \\ T(n) &= \Theta(n) + T(n-1) = \Theta(n^2) \end{aligned}$$

W każdym kroku iteracji pamiętamy dwie listy, na których razem jest n elementów. Ze względu na rekurencję ogonową, uzyskujemy złożoność pamięciową rzędu $M(n) = \Theta(n)$. Biorąc pod uwagę wielkość wyniku, jest to złożoność optymalna.

10.2 Insertion sort

Podział: posortowanie krótszego ciągu + wstawienie jednego elementu do posortowanej listy.

```
let wstaw l x =
  (filter (fun y ->y <= x) l) @
  (x :: (filter (fun y ->y >x) l));;

let insertion_sort l = fold_left wstaw [] l;;
```

Złożoność czasowa `wstaw` jest liniowa ze względu na długość przetwarzanej listy, a złożoność czasowa `insertion_sort` jest rzędu:

$$\begin{aligned} T(0) &= \Theta(1) \\ T(n) &= T(n-1) + \Theta(n) = \Theta(n^2) \end{aligned}$$

Złożoność pamięciowa jest rzędu $M(n) = \Theta(n)$, gdyż w każdej chwili pamiętamy tylko kilka list o łącznej długości rzędu $\Theta(n)$.

Można też poprawić trochę złożoność tego algorytmu. *Inwersją* w ciągu $[x_1; x_2; \dots; x_n]$ nazywamy każdą taką parę indeksów $1 \leq i < j \leq n$, dla której $x_i > x_j$. Oznaczmy przez i liczbę inwersji w danym ciągu do posortowania. Wszystkich możliwych par indeksów do rozważenia jest $\frac{n(n-1)}{2}$. Jeżeli dany ciąg jest malejący, to $i = \frac{n(n-1)}{2}$. Tak więc $i = \Theta(n^2)$. Z drugiej strony, dla ciągu rosnącego mamy $i = 0$.

Przyjrzyjmy się następującej wersji sortowania przez wstawianie:

```
let wstaw x l =
  let rec wst a x l =
    match l with
    [] -> rev(x::a) |
    h::t ->
      if x > h then wst (h::a) x t
      else rev (x::a) @ l
  in wst [] x l;;

let insertion_sort l = fold_right wstaw l [];;
```

W każdym kroku iteracji w procedurze `wst` pozbywamy się jednej inwersji związanej z elementem x . Stosując taki algorytm wstawiania otrzymujemy sortowanie działające w czasie $\Theta(n+i)$. Oczywiście liczba inwersji może być rzędu $\Theta(n^2)$. W następnym punkcie zobaczymy zastosowanie takiej implementacji sortowania przez wstawianie.

10.3 Sortowanie Shella

She sells sea shells on the sea shore;

The shells that she sells are sorted I'm sure.

She shall sort sea shells with a Shell-sort, though. So that the shells are Shell-sorted sea shore shells.

... Sortowanie Shella to uogólnienie sortowania przez wstawianie. Składa się on z wielu faz. W każdej fazie dzielimy ciąg na k krótszych ciągów, każdy złożony z co k -tych elementów. Na przykład dla $k = 2$ mamy dwa ciągi, jeden złożony z elementów na parzystych pozycjach i jeden złożony z elementów na nieparzystych elementach. Każdy z k ciągów sortujemy niezależnie, za pomocą sortowania przez wstawianie.

Na fazę sortowania można też spojrzeć tak: Elementy sortowanego ciągu wpisujemy wierszami do tabeli o k kolumnach, następnie każdą kolumnę sortujemy za pomocą sortowania przez wstawianie, po czym odczytujemy elementy wierszami.

W rezultacie uzyskujemy ciąg (x_1, \dots, x_n) , który jest k -posortowany, tzn. $x_i \leq x_{i+k}$ dla $1 \leq i \leq n - k$. Naszym celem jest uzyskanie ciągu posortowanego, czyli 1-posortowanego. W kolejnych fazach parametr k jest coraz mniejszy, aż w ostatniej fazie mamy $k = 1$. Tym samym oczywiste jest, że uzyskujemy ciąg posortowany. Po co więc poprzedzające fazy? Koszt sortowania przez wstawianie jest niewielki, jeżeli sortujemy ciąg, który jest już „prawie posortowany”. Idea sortowania Shella polega na tym, że w każdej kolejnej fazie sortujemy przez wstawianie właśnie takie ciągi, które są „prawie posortowane”.

Pozostaje pytanie, jaki powinien być ciąg wartości k ? Jest to skomplikowany problem i nie wszystko wiadomo o związkach między ciągiem wartości k , a złożonością czasową algorytmu Shella. Pożądane jest, aby ciąg ten miał następującą własność: jeśli ciąg posortowany co k posortujemy co k' , to nadal pozostanie on posortowany co k . Dobre znane ciągi, to:

- $\dots, 121, 40, 13, 4, 1$ — $k_i = 3h_{k+1} + 1$,
- $\dots, 31, 15, 7, 3, 1$ — $k_i = 2h_{k+1} + 1$.

Dla tego drugiego ciągu algorytm Shella ma złożoność czasową $\Theta(n^{1.5})$.

[N.Wirth, Algorytmy + Struktury danych = Programy]

[D.Knuth, Sztuka programowania] [[Implementacja]]

10.4 Sortowanie przez scalanie

Idea algorytmu — podziel listę, posortuj powstałe listy i scal je. Realizacja:

```
let rec merge_sort l =
  let split l =
    fold_left (fun (l1,l2) x ->(l2,x::l1)) ([],[]) l
  and merge l1 l2 =
    let rec mrg a l1 l2 =
      match l1 with
      [] ->(rev a) @ l2 |
      (h1::t1) ->
        match l2 with
        [] ->(rev a) @ l1 |
```

```

        (h2::t2) ->
            if h1 >h2 then
                mrg (h2::a) l1 t2
            else
                mrg (h1::a) t1 l2
    in
        mrg [] l1 l2
in
    match l with
    [] ->[] |
    [x] ->[x] |
    _ ->
        let
            (l1,l2) = split l
        in
            merge (merge_sort l1) (merge_sort l2);;

```

Analiza złożoności.

Czas: Każda faza wymaga liniowego czasu plus dwa razy wywołuje fazy dla swoich połówek.

$$\begin{aligned}
 T(1) &= 1 \\
 T(n) &= \Theta(2n) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)
 \end{aligned}$$

Czyli dla $n = 2^k$ mamy:

$$T(n) = \Theta(2n) + 2T\left(\frac{n}{2}\right) = \sum_{i=0}^k 2n = 2(k+1)n = \Theta(n \log n)$$

Można pokazać, że $T(n)$ jest monotoniczna, a więc $T(n) = \Theta(n \log n)$ dla dowolnych n , co jest optymalne.

Pamięć: Rozważmy ile pamięci jest zajętej przy największym zagłębieniu rekurencyjnym. Mamy listę do posortowania (n), jej dwie połówki, dla drugiej wywołanie rekurencyjne ($\frac{n}{2}$), pierwsza połówka jest już posortowana ($\frac{n}{2}$), a druga jeszcze nie. Analogicznie, jedna z połówek jest podzielona na dwie ćwiartki, a jedna z ćwiartek jest już posortowana, itd. W efekcie uzyskujemy optymalną złożoność pamięciową:

$$M(n) = 1.5n + M(n/2) \leq 1.5 \sum_{k \geq 0} \frac{n}{2^k} = \Theta(n)$$

Ćwiczenia

1. Tomek ma zabawkę, z której wystają drewniane słupki różnej wysokości. Jednym uderzeniem młotka może wbić lub wysunąć wybrany słupek o 1.

Napisz procedurę `słupki`, która dla danej listy początkowych wysokości słupków obliczy minimalną liczbę uderzeń młotka potrzebnych do wyrównania wysokości słupków.

2. Dysponujemy pewną pulą wolnych bloków pamięci. W tych blokach musimy umieścić pewien zestaw zmiennych, przy czym:

- w jednym bloku może znajdować się co najwyżej jedna zmienna,
- blok, w którym znajduje się dana zmienna nie może być od niej mniejszy.

Napisz procedurę `da_się`: `int list → int list → bool`, która na podstawie listy wielkości wolnych bloków oraz listy wielkości zmiennych ustali, czy da się rozmieścić zmienne w blokach.

3. Napisz funkcję `elementy`: `α list → int list → α list`, która dla list $[x_1; x_2; \dots; x_n]$ i $[y_1; y_2; \dots; y_m]$ zwraca listę $[x_{y_1}; x_{y_2}; \dots; x_{y_m}]$. Możesz założyć, że liczby całkowite y_i w drugiej liście są z przedziału $[1, n]$, gdzie n oznacza długość pierwszej listy.
4. [II OI] Napisz procedurę `trójką`: `int list → bool`, która dla danej listy $[x_1; x_2; \dots; x_n]$ dodatnich liczb całkowitych sprawdzi, czy lista zawiera trójkę elementów x_i, x_j i x_k (dla $i \neq j, j \neq k, i \neq k$) spełniających nierówność trójkąta, tzn.:

$$2 \cdot \max(x_i, x_j, x_k) \leq x_i + x_j + x_k$$

Podaj złożoność czasową i pamięciową swojego rozwiązania. (Uwaga: Jeżeli liczby całkowite mają ograniczony zakres, to można to zadanie rozwiązać w stałym czasie.)

5. Dana jest (niepusta) lista $[x_1; x_2; \dots; x_n]$ oraz liczba d . Szukamy takich elementów x_i i x_j , dla których wartość $|x_i - x_j|$ jest jak najbliższa d (tzn. wartość $||x_i - x_j| - d|$ jest minimalna).

Napisz procedurę `różnica`: `int list → int → int`, która dla danej listy oraz wartości d wyznacza różnicę elementów $|x_i - x_j|$ najbliższą d .

Podaj złożoność czasową i pamięciową swojego rozwiązania.

6. Napisz procedurę `przedział`: `int list → int → int`, która dla danej listy $[x_1; \dots; x_n]$ oraz dla liczby całkowitej $r \geq 0$ obliczy taką liczbę całkowitą c , że $|\{i : |x_i - c| \leq r\}|$ jest maksymalne.

Przykład: `przedział [2; -2; 5; -1; 11; 8; 4; 5; 8; 7] 2 = 6`.

7. Dana jest lista liczb rzeczywistych $[x_1; x_2; \dots; x_n]$. Napisz procedurę `przekładaniec`, której wynikiem jest taka permutacja $[x_{p_1}; x_{p_2}; \dots; x_{p_n}]$ danej listy, dla której suma

$$\sum_{i=1}^{n-1} |x_{p_{i+1}} - x_{p_i}|$$

jest największa.

8. System -2 -kowy jest podobny do systemu binarnego, ale podstawą tego systemu jest liczba -2 . Są dwie możliwe cyfry: 0 i 1 . Ciąg cyfr postaci $c_k c_{k-1} \dots c_1 c_0$ reprezentuje liczbę $\sum_{i=0}^k c_i (-2)^i$. Na przykład, liczbę 42 reprezentujemy jako 1111110_{-2} . Przyjmujemy, że lista pusta reprezentuje 0 .
- (a) Napisz procedurę `inc : int list → int list`, która dla danej listy będącej reprezentacją liczby x , wyznaczy reprezentację liczby $x + 1$.
- (b) Napisz procedurę `dodawanie : int list → int list → int list`, która dla danych dwóch reprezentacji liczb całkowitych w systemie -2 -ym (zawierających cyfry w kolejności od mniej do bardziej znaczących) obliczy reprezentację ich sumy. Na przykład: `dodawanie [1;0;1;0;0;1;1] [1;0;1] = [0;1;1;1;1;1;1]`.
9. Dana jest lista liczb całkowitych $[x_1; \dots; x_n]$. Napisz procedurę `pierwsze : int list → int list`, która zwróci taką jej **spójną** podlistę, że:
- każde dwa elementy podlisty są względnie pierwsze,
 - zawiera ona maksymalną możliwą liczbę elementów.

Wytyczne dla prowadzących ćwiczenia

Ad. 1 Do wyznaczenia mediany należy użyć sortowania.

Wykład 11. Zasada „dziel i rządź” i analiza złożoności na przykładzie algorytmów sortowania, c.d.

11.1 Quick-sort — Analiza oczekiwanej złożoności czasowej

Algorytm quick-sort opiera się na następującym pomysłe: dzielimy elementy ciągu względem wybranego elementu (pierwszego lub losowego) na mniejsze i większe, następnie sortujemy je niezależnie i sklejamy.

```
let rec quick_sort l =
  let split l s = (filter (fun y ->y <s) l,
                  filter (fun y ->y = s) l,
                  filter (fun y ->y >s) l)
  in
  if length l <2 then l else
    let s = nth l (Random.int (length l))
    in
      let (l1,l2,l3) = split l s
      in
        (quick_sort l1) @ l2 @ (quick_sort l3);;
```

Pesymistyczna złożoność tego algorytmu nie jest najlepsza, gdyż nie jest on odporny na wybór elementów w porządku rosnącym. W takim przypadku:

$$M(0) = 1$$

$$M(n) = \Theta(n) + M(n-1) = \Theta(n^2)$$

$$T(0) = 1$$

$$T(n) = \Theta(n) + T(n-1) = \Theta(n^2)$$

Zbadajmy jednak jak sprawa wygląda średnio.

Tw. 3. Algorytm Quick-sort, dla permutacji zbioru $\{1, \dots, n\}$ ma oczekiwaną złożoność czasową $\Theta(n \log n)$, a pamięciową $\Theta(n)$.

Dowód. Oznaczmy przez $T(n)$ oczekiwany czas działania algorytmu quick-sort, a przez $M(n)$ oczekiwany koszt pamięciowy. Korzystając z lematu możemy sformułować równanie rekurencyjne określające rząd $T(n)$. Każda z możliwych wartości s jest tak samo prawdopodobna, stąd:

$$T(n) = \frac{1}{n} \left(\sum_{s=1}^n (T(s-1) + T(n-s) + \Theta(n)) \right) = \quad (1)$$

$$= \frac{2}{n} \left(\sum_{s=0}^{n-1} T(s) \right) + \Theta(n) \quad (2)$$

$$T(0) = T(1) = \Theta(1) \quad (3)$$

Dla uproszczenia dalszych przekształceń dopuścimy się pewnego nadużycia notacji $\Theta(\dots)$ opartego na następującej obserwacji. W zależności od tego, czy chcemy oszacować rząd $T(n)$

od góry, czy od dołu, w miejsce $\Theta(n)$ możemy wstawić $a \cdot n$, gdzie a jest odpowiednio dobraną stałą $a > 0$. Analogicznie, w miejsce $\Theta(1)$ możemy wstawić odpowiednio dobraną stałą $b > 0$. Dalsze przekształcenia będziemy prowadzić tak, jakby w miejsce $\Theta(\dots)$ zostały wstawione odpowiednie wyrażenia, myśląc równocześnie o oszacowaniu rzędu od góry i od dołu. Nie będziemy jednak powtarzać tych samych przekształceń dwukrotnie, raz dla oszacowania rzędu od góry, a raz od dołu. Korzystając z (4) oszacujemy $nT(n)$:

$$\begin{aligned}
 nT(n) &= 2 \sum_{s=0}^{n-1} T(s) + nf(n) = \\
 &= 2 \sum_{s=0}^{n-1} T(s) + nf(n) + (n-1)T(n-1) - \\
 &\quad - 2 \sum_{s=0}^{n-2} T(s) - (n-1)f(n-1) = \\
 &= 2T(n-1) + (n-1)T(n-1) + \Theta(n) = \\
 &= (n+1)T(n-1) + \Theta(n)
 \end{aligned}$$

Dzieląc obie strony równania przez $n(n+1)$ otrzymujemy:

$$\begin{aligned}
 \frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \Theta\left(\frac{1}{n+1}\right) = \\
 &= \Theta\left(\frac{1}{n+1} + \frac{1}{n} + \dots + \frac{1}{3}\right) + \Theta(1) = \\
 &= \Theta(\log n)
 \end{aligned}$$

Stąd $T(n) = \Theta(n \log n)$.

Koszt pamięciowy liczymy podobnie. Najwięcej pamięci potrzebujemy, gdy (na każdym poziomie rekurencji) pierwsze wywołanie rekurencyjne zakończyło się, a drugie jest w trakcie.

$$\begin{aligned}
 M(n) &= \frac{1}{n} \left(\sum_{s=1}^n (\Theta(s-1) + M(n-s) + \Theta(n)) \right) = \\
 &= \frac{1}{n} \left(\sum_{s=0}^{n-1} M(s) \right) + \Theta(n) \\
 M(0) &= M(1) = \Theta(1)
 \end{aligned}$$

Stąd:

$$\begin{aligned}
 nM(n) &= \sum_{s=0}^{n-1} M(s) + nf(n) = \\
 &= \sum_{s=0}^{n-1} M(s) + nf(n) + (n-1)M(n-1) - \\
 &\quad - \sum_{s=0}^{n-2} M(s) - (n-1)f(n-1) = \\
 &= M(n-1) + (n-1)M(n-1) + \Theta(n) = \\
 &= (n+1)T(n-1) + \Theta(n) \\
 \frac{M(n)}{n+1} &= \frac{M(n-1)}{n} + \Theta\left(\frac{1}{n+1}\right) = \\
 &= \Theta\left(\frac{1}{n+1} + \frac{1}{n} + \dots + \frac{1}{3}\right) + \Theta(1) = \\
 &= \Theta(\log n)
 \end{aligned}$$

Stąd $T(n) = \Theta(n \log n)$.

□

11.2 Heap-sort

11.2.1 Kolejka priorytetowa

Kolejka priorytetowa to kolekcja elementów ze zbioru z określonym porządkiem liniowym, na której mamy określone następujące operacje:

```

module type PRI_QUEUE = sig
  type 'a pri_queue                                     typ kolejek
  val empty_queue : 'a pri_queue                       pusta kolejka
  val is_empty : 'a pri_queue -> bool                 czy pusta?
  val put : 'a pri_queue ->'a ->'a pri_queue         włożenie elementu
  val getmax : 'a pri_queue ->'a                     maximum
  val removemax : 'a pri_queue ->'a pri_queue       usuń maximum
  exception Empty_Queue                                gdy kolejka pusta
end;;

```

Wartości (abstrakcyjne) tego typu możemy sobie przedstawić jako zbiory (lub lepiej multi-zbiory) wartości elementów.

11.2.2 Sortowanie

Posłużmy się zasadą pobożnych życzeń. Załóżmy, że mamy dostępną *kolejkę priorytetową*. Mając dostępną taką kolejkę możemy zrealizować sortowanie, najpierw wkładając wszystkie elementy do kolejki, a potem wyjmując je w porządku nierosnącym.

```

let heap_sort l =
  let wloz = fold_left put empty_queue l
  and wyjmij (l,q) = ((getmax q)::l,removemax q)
  in
    iteruj
      ([],wloz)
      wyjmij
      (fun (_,q) ->is_empty q)
      (fun (l,_) ->l)

```

Złożoność czasowa tego algorytmu zależy oczywiście od implementacji kolejki priorytetowej. Mamy tutaj n operacji włożenia do kolejki i n operacji wyjęcia maksimum. Pozostały koszt jest rzędu $\Theta(n)$.

Oszacowanie kosztu pamięciowego jest prostsze. Możemy założyć, że kolejka zawierająca n elementów zajmuje $\Theta(n)$ pamięci (a przynajmniej tak będzie dla wszystkich rozpatrywanych dalej implementacji). Przy takim założeniu złożoność pamięciowa jest rzędu $\Theta(n)$ i jest optymalna, ze względu na rozmiar wyniku.

11.2.3 Implementacja listowa kolejek — selection sort

Możliwych jest wiele realizacji kolejek priorytetowych. Zaczniemy od nieefektywnej, ale prostej realizacji listowej. Elementy kolejki pamiętamy na liście, w dowolnej kolejności. Funkcja abstrakcji przyporządkowuje liście multizbiór jej elementów.

```

module Unordered_Pri_Queue : PRI_QUEUE = struct
  exception Empty_Queue
  type 'a pri_queue = 'a list
  let empty_queue = []
  let is_empty q = q = []
  let put q x = x::q
  let getmax q =
    if q = [] then raise Empty_Queue
    else fst (select_max q)
  let removemax q =
    if q = [] then raise Empty_Queue
    else snd (select_max q)
end;;

```

Algorytm sortowania oparty na takiej realizacji kolejki, to sortowanie przez wybieranie, $T(n) = \Theta(n^2)$.

11.2.4 Implementacja listowa kolejek — insertion sort

Możemy też zastosować inną implementację listową kolejki priorytetowej. Konkretnie wartości kolejek, to dowolne listy o elementach uporządkowanych nierosnąco. Funkcja abstrakcji, podobnie jak poprzednio, przyporządkowuje kolejce multizbiór jej elementów. Jednak tym razem jest to funkcja 1–1.

```

module Ordered_Pri_Queue : PRI_QUEUE = struct
  exception Empty_Queue
  type 'a pri_queue = 'a list
  let empty_queue = []
  let is_empty q = q = []
  let put q x =
    (filter (fun y ->y >x) q) @
    (x :: (filter (fun y ->y <= x) q))
  let getmax q =
    if q = [] then raise Empty_Queue
    else hd q
  let removemax q =
    if q = [] then raise Empty_Queue
    else tl q
end;;

```

Algorytm sortowania oparty na takiej realizacji kolejki, to sortowanie przez wstawianie.

11.2.5 Stóg

Kolejkę priorytetową można pamiętać efektywniej, w postaci tzw. stogu. Stóg jest to drzewo binarne, w którego węzłach są przechowywane liczby i w którym dla każdego poddrzewa spełniony jest następujący warunek: wartość w korzeniu poddrzewa jest większa lub równa wszystkim wartościom przechowywanym w poddrzewie. Dodatkowo dla każdego poddrzewa w jego korzeniu pamiętamy jego wielkość. Stóg i operacje na nim możemy zaimplementować w następujący sposób:

```

module Heap_Pri_Queue : PRI_QUEUE = struct
  exception Empty_Queue

  type 'a pri_queue =
    Node of 'a * 'a pri_queue * 'a pri_queue * int |
    Null

  let empty_queue = Null

  let is_empty q = q = Null

  let size q =
    match q with
    | Null -> 0 |
    | Node (_,_,_,n) -> n

  let getmax h =
    match h with
    | Null -> raise Empty_Queue |
    | Node (r,_,_,_) -> r

  let set_root h r =
    match h with

```

```

Null ->Node (r,Null,Null,1) |
Node (_,l,p,n) ->Node (r,l,p,n)

let rec put h x =
  match h with
  Null ->Node (x,Null,Null,1) |
  Node (r,l,p,n) ->
    if size l <= size p then
      Node((max x r),(put l (min x r)),p,(n+1))
    else
      Node((max x r),l,(put p (min x r)),(n+1))

let rec removemax h =
  match h with
  Null ->raise Empty_Queue |
  Node (_,Null,Null,_) ->Null |
  Node (_,l,Null,_) ->l |
  Node (_,Null,p,_) ->p |
  Node (_,(Node (r1,_,_,_) as l),
        (Node (rp,_,_,_) as p),n) ->
    if r1 >= rp then
      Node (r1,removemax l,p,n - 1)
    else
      Node (rp,l,removemax p,n - 1)
end;;

```

Fakt 3. Załóżmy, że dokonujemy n operacji `put` i n operacji `getmax`. Wysokość drzewa stogu nie przekracza $\lceil \log_2 n \rceil$.

Dowód. Zauważmy, że jeśli stóg nie jest drzewem pełnym, to wstawienie nie zwiększa jego wysokości. Gdyby były tylko operacje wstawiania lub najpierw wstawialibyśmy elementy, a potem je wyjmowali, to stóg nie miałby przez to mniejszej wysokości. Wystarczy więc rozważyć przypadek, gdy najpierw mamy operacje `put`, a potem `getmax`.

W trakcie fazy wstawiania, dla każdego węzła zachodzi następujący warunek: liczba wierzchołków w lewym i prawym poddrzewie różni się o co najwyżej 1. Wynika stąd, że w fazie wstawiania stóg jest drzewem zrównoważonym. Z faktu 2, który mówi, że drzewo binarne o n wierzchołkach ma wysokość $\lceil \log_2 n \rceil$, wynika teza. \square

Zauważmy, że koszt czasowy operacji wstawiania jest proporcjonalny do $\log_2 k$, gdzie k jest aktualną liczbą elementów w drzewie, czyli wynosi $\Theta(\log k) = O(\log n)$. Natomiast koszt (czasowy i pamięciowy) usuwania jest co najwyżej proporcjonalny do wysokości drzewa, czyli jest $O(\log n)$. Stąd koszt czasowy sortowania za pomocą stogu wynosi $\Theta(n \log n)$. Pamięciowy jest oczywiście $\Theta(n)$.

11.3 Koszt zamortyzowany

Przy okazji stogu zastanówmy się nad takim problemem: Mamy ciąg n operacji wstawień i usunięć elementów ze stogu. Zwykle rozmiar stogu jest dużo mniejszy niż n . Czy potrafimy lepiej oszacować koszt wstawiania i usuwania?

Koszt wstawiania jest rzędu $\Theta(\log k)$, gdzie k to aktualny rozmiar stogu, gdyż rozmiar drzewa, do którego wstawiamy, w każdym kroku maleje przynajmniej dwukrotnie. Koszt

usuwania jest co najwyżej proporcjonalny do wysokości stogu. Zauważmy, że wysokość stogu może być istotnie większa niż $\log_2 k$. Okazuje się, że możemy „sumarycznie” przyjąć, że:

- koszt operacji wstawiania jest rzędu $\Theta(\log k)$,
- koszt operacji usuwania jest stały.

To drugie stwierdzenie jest trochę dziwne. Co to jednak znaczy „sumarycznie”? To znaczy, że jeżeli policzymy sumę kosztów wszystkich operacji w ciągu, zgodnie z tą zasadą, to wynik będzie poprawny.

Wyobraźmy sobie, że wykonywanie obliczeń wymaga energii (proporcjonalnej do czasu ich trwania), a my staramy się oszacować ilość zużytej energii. Jednak w trakcie wkładania elementów do stogu zużywamy energię nie tylko na obliczenia. Wstawiany element musi być umieszczony odpowiednio wysoko w stogu. Musimy go więc podnieść na odpowiednią wysokość, a tym samym nadać mu pewną energię potencjalną proporcjonalną do jego wysokości. Tak się składa, że wstawiając element, wykonujemy dokładnie tyle samo kroków, co wysokość, na jakiej element jest umieszczany, a więc zużywamy energię rzędu $O(\log_2 n)$. Przy tym, wszelkie niezrównoważenia w kształcie stogu mogą ten koszt tylko zmniejszyć.

Z kolei, gdy wyjmujemy element ze stogu, to wykonujemy szereg rotacji. Każda rotacja wykonuje się w stałym czasie i powoduje obniżenie jednego elementu w stogu o jedną jednostkę wysokości. Możemy więc pokryć energię potrzebną do wykonania obliczeń energią potencjalną obniżanego elementu. Tak więc usuwanie, choć trochę może potrwać, będzie wymagało (co najwyżej) stałej ilości energii.

Oczywiście nasz ciąg nie musi usuwać ze stogu wszystkich elementów, więc część zużytej przez nas energii może pozostać zmagazynowana w stogu. Energia pustego stogu jest równa zero, a energia każdego innego stogu jest dodatnia. Tak więc nie „pożyczamy” znikąd energii, a co najwyżej trochę jej pozostawiamy zmagazynowanej w stogu. A zatem, łączny czas obliczeń jest *co najwyżej* taki, jak ilość zużytej energii.

Bardziej formalnie, postępujemy następująco:

- określamy *funkcję potencjału* zależną od wartości naszej struktury danych,
- jeśli w kolejnym kroku przechodzimy do nowej wartości struktury danych, to doliczamy do kosztu kroku różnicę nowej i starej wartości funkcji potencjału,
- po zakończeniu obliczeń możemy od kosztu odjąć różnicę między wartością funkcji potencjału dla końcowej i początkowej wartości struktury danych.

11.4 [[Plakaty, XV OI]]

11.5 Kolejka FIFO — przykład kosztu zamortyzowanego

Kolejka FIFO to taka kolejka, z której wyjmujemy elementy w takiej kolejności, w jakiej były wstawiane. Interfejs kolejki możemy ująć w postaci takiej oto sygnatury:

```
module type QUEUE =
  sig
    exception EmptyQueue
    type 'a queue
    val empty : 'a queue
```

```

val is_empty : 'a queue ->bool
val insert : 'a queue ->'a ->'a queue
val front : 'a queue ->'a
val remove : 'a queue ->'a queue
end;;

```

Jak zaimplementować kolejkę, dla której operacje mają koszt stały? Można uzyskać koszt stały zamortyzowany.

Elementy kolejki przechowujemy na dwóch listach. Do jednej dokładamy elementy, a z drugiej wyjmujemy. W momencie, gdy kolejka elementów do wyjmowania jest pusta, to „przelewamy” elementy z jednej kolejki do drugiej, odwracając ich kolejność. Dodatkowo pamiętamy rozmiar kolejki i pierwszy jej element.

```

module Fifo : QUEUE =
  struct
    exception EmptyQueue

    (* Kolejka to trójka: przód,tył,rozmiar. *)
    (* Jeżeli przód jest pusty,to kolejka jest pusta. *)
    type 'a queue = {front: 'a list; back: 'a list; size: int}

    let empty = {front=[]; back=[]; size=0}

    let size q = q.size

    let is_empty q = size q = 0

    let balance q =
      match q with
      {front=[]; back=[]}      ->q |
      {front=[]; back=b; size=s} ->{front=rev b; back=[]; size=s} |
      -                        ->q

    let insert {front=f; back=b; size=n} x =
      balance {front=f; back=x::b; size=n+1}

    let front q =
      match q with
      {front=[]}  ->raise EmptyQueue |
      {front=x::_} ->x

    let remove q =
      match q with
      {front=::_:f} ->balance {front=f; back=q.back; size=q.size-1} |
      -             ->raise EmptyQueue
  end;;

```

Zanalizujmy koszt zamortyzowany operacji. Funkcja potencjału to `length q.back`. Procedury `is_empty_queue`, `size` i `front` mają koszt stały i nie zmieniają funkcji potencjału. Dla pustej kolejki funkcja potencjału jest równa 0. Procedura `insert` ma koszt stały i powoduje zwiększenie funkcji potencjału o 1, co daje stały koszt zamortyzowany. Procedura `remove`

może mieć koszt liniowy, ze względu na procedurę `balance`. Jednak ta ostatnia wykonuje tyle kroków, o ile zmniejsza funkcję potencjału. Tak więc koszt zamortyzowany procedury `remove` jest stały.

11.6 Deser

```
int getRandomNumber()  
{  
    return 4; // chosen by fair dice roll.  
             // guaranteed to be random.  
}
```

<http://xkcd.com/221/>

Ćwiczenia

1. Sformułuj warunek określający, czy element wstawiany do drzewa ma być wstawiony do lewego, czy prawego poddrzewa, tak aby kolejne elementy były wstawiane na kolejnych poziomach od lewej do prawej.
2. Napisz funkcję, która przekształci zadane drzewo w stóg, zachowując jego kształt (w rozwiązaniu można wykorzystać zmodyfikowaną procedurę `rotate` z wykładu). Jaka jest złożoność tej procedury? W jaki sposób zależy ona od kształtu drzewa?
3. Napisz funkcję, która „wyważy” zadany stóg, to znaczy tak poprzestawia w nim elementy, aby miał minimalną wysokość. Jaka jest jej złożoność?
4. Rozszerzyć implementację kolejek FIFO o wkładanie i wyjmowanie elementów z obydwu stron (w koszcie zamortyzowanym stałym).
5. Mamy n kubeczków z wodą, ponumerowanych od 1 do n ($n > 0$). W kubeczku nr i znajduje się x_i wody, przy czym $0 < x_1 \leq x_2 \leq \dots \leq x_n$. Powtarzamy k razy (dla $k < n$) następującą czynność: wybieramy dwa (niepuste) kubeczki zawierające jak najmniej wody i wodę z jednego kubeczka dolewamy do drugiego.

Napisz procedurę `zlewki` : `float list` \rightarrow `int` \rightarrow `float`, która dla danej listy zawierającej liczby $[x_1; \dots; x_n]$ oraz liczby k określa ile jest wody w najmniej wypełnionym niepustym kubeczku po wykonaniu k operacji przelewania.

6. Zadanie o katastrofach lotniczych: Dana jest lista liczb całkowitych $[x_1; \dots; x_n]$. Dla każdego i trzeba policzyć największe takie k_i , że $x_i = \max(x_{i-k_i+1}, \dots, x_i)$. Przyjmujemy, że $x_0 = \infty$.
7. Zaimplementuj k -max-kolejkę, czyli kolejkę, do której można wkładać elementy i dowiadywać się jakie jest maksimum z k ostatnio włożonych elementów. Dokładniej, powinny być dostępne następujące operacje na k -max-kolejkach:

- `init` : `int` \rightarrow `α max_queue` — tworzy nową pustą kolejkę z określonym parametrem k ,
- `put` : `α max_queue` \rightarrow `α` \rightarrow `α max_queue` — wkłada element do kolejki,
- `get_max` : `α max_queue` \rightarrow `α` — zwraca maksimum z k ostatnich elementów włożonych do kolejki.

Wszystkie operacje na kolejce powinny działać w stałym czasie zamortyzowanym.

8. Dana jest lista $[x_1; \dots; x_n]$ i stała $0 \leq k < n$. Oblicz listę $[y_1; \dots; y_n]$, gdzie $y_i = \max(x_{\max(i-k,1)}, \dots, x_i)$.
9. Mamy daną listę liczb całkowitych $l = [a_1, \dots, a_n]$. Liczbę a_i nazywamy k -maksimum, jeżeli jest ona większa od $a_{\max(i-k,1)}, \dots, a_{i-1}, a_{i+1}, a_{\min(i+k,n)}$. Napisz funkcję `kmax`: `int` \rightarrow `int list` \rightarrow `int list`, dla której wynikiem `kmax` k l jest podlista listy l złożona z samych k -maksimów.
10. Dana jest lista $[x_1; \dots; x_n]$. Dla $1 \leq i < j \leq n$ powiemy, że elementy x_i i x_j widzą się nawzajem, jeżeli $\min(x_i, x_j) \geq \max(x_{i+1}, \dots, x_{j-1})$.

Napisz procedurę `widoczne`, która obliczy ile elementów ciągu jest widocznych z kolejnych jego elementów. Np. `widoczne [1; 8; 5; 6; 4] = [1; 3; 2; 3; 1]`.

11. Dana jest tablica n dodatnich liczb całkowitych. Tablica ta opisuje figurę złożoną z n stykających się bokami prostokątów-słupków, których dolne boki leżą na jednej prostej (tzw. Manhattan skyline). Kolejne liczby w tej tablicy reprezentują wysokości kolejnych słupków jednostkowej szerokości (od lewej do prawej).

Napisz procedurę `prostokat`: `int list` \rightarrow `int`, która dla danej tablicy wyznaczy maksymalną powierzchnię prostokąta (o bokach równoległych do boków figury), który można wpisać w figurę opisaną przez tablicę.

12. Dana jest lista liczb całkowitych $[x_1; \dots; x_n]$. Dla każdego i trzeba policzyć największe takie otoczenie x_i , czyli podciąg $x_{i-k_i}, \dots, x_{i+k_i}$, że $x_i = \max(x_{i-k_i}, \dots, x_{i+k_i})$.

13. [PCh] Dana jest lista liczb całkowitych $[x_1; \dots; x_n]$. Napisz procedurę `różnica`: `int list` \rightarrow (`int * int`), która dla danej listy wyznaczy taką parę liczb (i, j) , że:

- $0 < i < j \leq n$,
- $x_i \leq x_j$, oraz
- $j - i$ jest jak największe.

Jeżeli takie i i j nie istnieją, to poprawnym wynikiem jest $(0, 0)$.

14. [SICP, Ćw 2.29] Mobil, to ruchoma konstrukcja o budowie rekurencyjnej. Mobil ma ruchome ramiona, jak w wadze szalkowej, określonej długości. Na końcu każdego z ramion zawieszony jest ciężarek o określonej masie lub mniejszy mobil.

- Zaimplementuj strukturę danych reprezentującą mobil.
- Napisz procedurę obliczającą wagę mobila.
- Napisz procedurę sprawdzającą, czy mobil jest wyważony.

* [BOI 1999 — uproszczone] Napisz procedurę, która zrównoważy dany mobil tak, aby jego odważniki miały dodatnie wagi całkowite, a jego całkowita masa była minimalna.

15. [XII OI, zadanie Sumy Fibonacciego] System Zeckendorfa, to taki system, w którym liczba jest reprezentowana jako ciąg zer i jedynek, a kolejne cyfry mają takie znaczenie, jak kolejne liczby Fibonacciego $(1, 2, 3, 5, \dots)$. Ponadto, w reprezentacji liczby nie mogą pojawiać się dwie jedyneki obok siebie.

Liczby reprezentujemy jako listy zer i jedynek. Zaimplementuj dodawanie w systemie Zeckendorfa.

Wytyczne dla prowadzących ćwiczenia

Ad. 6 Wymaga użycia stosu.

Ad. 7 Rozwiązując to zadanie należy skorzystać z rozwiązania zadania 4 i zadania 6.

Ad. 8 Należy skorzystać z rozwiązania zadania 7.

Ad. 9 Należy skorzystać z rozwiązania zadania 8 lub bezpośrednio z 7.

Ad. 10 Rozszerz rozwiązanie zadania 6.

Ad. 12 Rozszerz rozwiązanie zadania 6.

Ad. 15 Stosunkowo łatwo można rozwiązać to zadanie w czasie $O(n^2)$, a dużo trudniej w czasie $O(n)$.

Wykład 12. Funktory

Funktory to moduły sparametryzowane. Często jeden moduł *importuje* inne moduły. Skoro rozdzieliliśmy pojęcia interfejsu (sygnatury) i implementacji (struktury), to możemy zapisywać moduły korzystające z innych modułów w bardziej abstrakcyjny sposób. Jeżeli spojrzymy na sygnaturę jak na specyfikację wykorzystywanego modułu, to zgodnie z zasadą ukrywania informacji (ang. *information hiding*), istotne powinny być jedynie specyfikacje wykorzystywanych modułów, a nie ich implementacje. Podstawienie dowolnej implementacji spełniającej zadaną specyfikację/sygnaturę powinno dać pożądane rezultaty. Moduł taki możemy zapisać w postaci sparametryzowanej — podajemy sygnatury parametrów, po podstawieniu w ich miejsce odpowiednich implementacji otrzymujemy wynikowy moduł.

Na funktory możemy też patrzeć jak na funkcje na strukturach — przetwarzają one moduły-parametry w moduły wynikowe. Tak jak procedury są jednym z możliwych typów wartości, tak funktory są rodzajem struktur. Możemy też mówić o sygnaturach funktorów. Pod względem składniowym, sposób definiowania funktorów i ich sygnatur przypomina sposób definiowania procedur i λ -abstrakcji:

$$\begin{aligned} \langle \text{sygnatura} \rangle &::= \text{functor } \langle \text{Identyfikator} \rangle _ \langle \text{sygnatura} \rangle _ _ _ \langle \text{sygnatura} \rangle \mid \dots \\ \langle \text{struktura} \rangle &::= \text{functor } \langle \text{Identyfikator} \rangle _ \langle \text{sygnatura} \rangle _ _ _ \langle \text{struktura} \rangle \mid \\ &\quad \langle \text{struktura} \rangle _ \langle \text{struktura} \rangle _ _ \mid \dots \\ \langle \text{definicja} \rangle &::= \text{module } \langle \text{Identyfikator} \rangle _ \langle \text{Identyfikator} \rangle _ \langle \text{sygnatura} \rangle _ \\ &\quad _ _ _ \langle \text{sygnatura} \rangle _ _ _ \langle \text{struktura} \rangle _ _ \dots \end{aligned}$$

Przykład: *Przestrzeń metryczna*, to zbiór punktów X , wraz z *metryką* $d : X \times X \rightarrow [0; \infty)$ określającą odległości między punktami. Metryka musi spełniać następujące warunki:

- $d(a, b) = 0 \Leftrightarrow a = b$,
- $d(a, b) = d(b, a)$,
- $d(a, b) + d(b, c) \geq d(a, c)$,

dla dowolnych $a, b, c \in X$.

Wyobraźmy sobie płaszczyznę \mathbb{R}^2 . Oto trzy przykłady najpopularniejszych metryk w tej przestrzeni:

- metryka miejska (Manhattan): $d_1((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_1 - y_2|$,
- metryka euklidesowa: $d_2((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$,
- metryka maksimum: $d_\infty((x_1, y_1), (x_2, y_2)) = \max(|x_1 - x_2|, |y_1 - y_2|)$,

Zdziwienia dnia:

- Jaki kształt mają okręgi w tych metrykach?
- Na płaszczyźnie metryki d_1 i d_∞ tworzą przestrzenie izomorficzne. Podaj przekształcenie układu współrzędnych, które przekształca jedną przestrzeń w drugą.

- Czy przestrzenie metryczne z metrykami d_1 i d_∞ też są izomorficzne? Jaki kształt mają sfery w tych przestrzeniach?

Zaimplementujmy pojęcie przestrzeni metrycznej oraz opisane trzy przestrzenie metryczne.

```

module type METRIC_SPACE = sig
  type t
  val d: t ->t ->float
end;;

module Manhattan_Plane : METRIC_SPACE = struct
  type t = float * float
  let d (x1,y1) (x2,y2) =
    abs_float (x1 -. x2) +. abs_float (y1 -. y2)
end;;

module Euclidian_Plane : METRIC_SPACE = struct
  type t = float * float
  let square x = x *. x
  let d (x1,y1) (x2,y2) =
    sqrt (square (x1 -. x2) +. square (y1 -. y2))
end;;

module Max_Plane : METRIC_SPACE = struct
  type t = float * float
  let d (x1,y1) (x2,y2) =
    max (abs_float (x1 -. x2)) (abs_float (y1 -. y2))
end;;

```

Możemy też zdefiniować, jako funktor, produkt kartezjański przestrzeni metrycznych. Musimy jednak określić w jaki sposób odległości w przestrzeniach składowych mają być przekształcane w odległość w przestrzeni wynikowej. Znowu, możemy to zrobić na trzy sposoby:

```

module Manhattan_Product (S1 : METRIC_SPACE) (S2 : METRIC_SPACE) : METRIC_SPACE =
struct
  type t = S1.t * S2.t
  let d (x1,y1) (x2,y2) = S1.d x1 x2 +. S2.d y1 y2
end;;

module Euclidian_Product (S1 : METRIC_SPACE) (S2 : METRIC_SPACE) : METRIC_SPACE =
struct
  type t = S1.t * S2.t
  let square x = x *. x
  let d (x1,y1) (x2,y2) = sqrt (square (S1.d x1 x2) +. square (S2.d y1 y2))
end;;

module Max_Product (S1 : METRIC_SPACE) (S2 : METRIC_SPACE) : METRIC_SPACE = struct
  type t = S1.t * S2.t
  let d (x1,y1) (x2,y2) = max (S1.d x1 x2) (S2.d y1 y2)
end;;

```

Jeśli zdefiniujemy trywialną przestrzeń metryczną \mathbb{R} z metryką $|\cdot|$, to nasze trzy przestrzenie metryczne na płaszczyźnie można też zdefiniować za pomocą powyższych funktorów.

```
module Line : METRIC_SPACE = struct
  type t = float
  let d x y = abs_float (x -. y)
end;;

module Manhattan_Plane = Manhattan_Product (Line) (Line);;
module Euclidian_Plane = Euclidian_Product (Line) (Line);;
module Max_Plane      = Max_Product (Line) (Line);;
```

Przykład: Kolejka priorytetowa.

Kolejka priorytetowa może być sparametryzowana porządkiem liniowym określonym na elementach. Porządek taki ma sygnaturę:

```
type porownanie = Mniejsze | Rowne | Wieksze;;

module type PORZADEK_LINIOWY =
  sig
    type t
    val porownaj : t ->t ->porownanie
  end;;
```

Kolejka priorytetowa powinna mieć sygnaturę postaci:

```
module type PRI_QUEUE_FUNC = functor (Elem : PORZADEK_LINIOWY) ->
  sig
    exception EmptyQueue
    type elem = Elem.t
    type queue
    val empty : queue
    val is_empty : queue ->bool
    val put : elem ->queue ->queue
    val getmax : queue ->elem
    val removemax : queue ->queue
  end;;
```

Zwróćmy uwagę na to, że typ `queue` jest abstrakcyjny, a `elem` nie. Ukrywamy w ten sposób strukturę kolejki. Natomiast cokolwiek będzie wiadome o typie `Elem.t` będzie nadal wiadome o typie `elem`. Jest to konieczne z tego powodu, że inaczej nie byłibyśmy w stanie włożyć żadnego elementu do kolejki. Implementację kolejki priorytetowej możemy zrealizować w następujący sposób:

```
module PriorityQueue : PRI_QUEUE_FUNC = functor (Elem : PORZADEK_LINIOWY) ->
  struct
    type elem = Elem.t
```

```

exception EmptyQueue
type queue = elem list
let empty = []
let is_empty q = q = empty
let rec put x q =
  if q = [] then [x]
  else if Elem.porownaj x (hd q) = Wieksze then x :: q
  else (hd q)::(put x (tl q))
let getmax q =
  if q = [] then raise EmptyQueue
  else hd q
let removemax (q:queue) =
  if q = [] then raise EmptyQueue
  else tl q
end;;

```

Kolejka priorytetowa liczb całkowitych może być zrealizowana w taki sposób:

```

module IntOrder =
  struct
    type t = int
    let porownaj (x:int) (y:int) =
      if x < y then Mniejsze else
      if x > y then Wieksze else Rowne
  end;;

module IntQueue = PriQueue (IntOrder);;

```

W module IntOrder typ t jest konkretny. Dzięki temu wiadomo, że elementy kolejki IntQueue.queue są typu int.

Porządek liniowy określa też kolejność sortowania:

```

module type SORTING = functor (Elem : PORZADEK_LINIOWY) ->
  sig
    type elem = Elem.t
    val sortuj : elem list ->elem list
  end;;

```

Jeden z algorytmów sortowania, to heap-sort.

```

module HeapSort (PrQ : PRI_QUEUE_FUNC): SORTING =
  functor (Elem : PORZADEK_LINIOWY) ->
    struct
      type elem = Elem.t
      module PQ = PrQ (Elem)
      open PQ
      let wkladaj l =
        List.fold_left (fun h x ->put x h) empty l
      let rec wyjmuj q a =
        if is_empty q then a
        else wyjmuj (removemax q) (getmax q :: a)
    end

```

```

    let sortuj l = wyjmuj (wkładaj l) []
  end;;

module Sortowanie = HeapSort (PriQueue);;

module S = Sortowanie (IntOrder);;

S.sortuj [1;7;3;6;5;2;4;8];;

```

Jak widać z powyższego przykładu, jeżeli będziemy zapisywać moduły w postaci funktorów, to będziemy z nich tworzyć łatwo wymienne elementy.

12.1 Funktory wyższych rzędów

Tak jak istnieją procedury wyższych rzędów, istnieją również funktory wyższych rzędów. Są to funktory, których parametrami i/lub wynikiem są funktory. Oto przykład, stanowiący kontynuację poprzedniego przykładu:

Przykład: Opierając się na sortowaniu możemy wprowadzić obliczanie mediany. Potrzebujemy do tego porządku liniowego i jakiegoś algorytmu sortowania.

```

module type MEDIANA =
  functor (Elem : PORZADEK_LINIOWY) ->
    sig
      type elem = Elem.t
      exception PustaLista
      val mediana : elem list ->elem
    end;;

module Mediana : functor (Sort : SORTING) ->MEDIANA =
  functor (Sort : SORTING) ->
    functor (Elem : PORZADEK_LINIOWY) ->
      struct
        type elem = Elem.t
        exception PustaLista
        module S = Sort (Elem)
        let mediana l =
          let s = S.sortuj l
            and n = List.length l
          in
            if n = 0 then
              raise PustaLista
            else
              List.nth s (n / 2)
        end;;
      end;;

module M = Mediana (HeapSort (PriQueue)) (IntOrder);;

M.mediana [4;3;5;6;2;1;7];;

```

12.2 Standardowe funktory — odpowiednik “szablonów”

[TODO: Rozwinąć]

- Standardowe biblioteki implementujące popularne struktury danych.
- Kontener wartości pasujących do pewnego interfejsu (np. porządku liniowego).
- Funktor — kompilowalny odpowiednik “szablonu”.
- Przykłady:
 - `Map` — słowniki o czasie dostępu $\Theta(n \log n)$.
 - `Set` — zbiory, czas dostępu $\Theta(n \log n)$.
- Więcej przykładów, gdy poznamy konstrukcje imperatywne.

12.2.1 `Map`

- `Map.Make` — funktor z porządku liniowego w strukturę słownikową.
- Sygnatura argumentu `Map.OrderedType`:

```
type t

val compare : t ->t ->int

mniejsze (< 0), równe (= 0), większe (> 0)
```

- Sygnatura wynikowa:

```
type key

type 'a t

val empty : 'a t

val add : key ->'a ->'a t ->'a t

val mem : key ->'a t ->bool

val find : key ->'a t ->'a

val remove : key ->'a t ->'a t

⋮
```

12.2.2 Set

- `Set.Make` — funktor z porządku liniowego w zbiory elementów.
- `Set.OrderedType = Map.OrderedType`
- Sygnatura wynikowa:

```
type elt

type t

val empty : t

val is_empty : t ->bool

val mem : elt ->t ->bool

val add : elt ->t ->t

val remove : elt ->t ->t

val union : t ->t ->t

val inter : t ->t ->t

val diff : t ->t ->t

val subset : t ->t ->bool

val equal : t ->t ->bool

:
```

12.3 Deser



<http://xkcd.com/1597/>

Ćwiczenia

- Monoid składania funkcji — funktor, który dla określonego typu \mathbf{t} konstruuje monoid z funkcją identycznościową i operacją składania funkcji (na \mathbf{t}).
- Zdefiniuj funktor, który dla danego monoidu definiuje operację potęgowania.
- Przypomnij sobie sygnaturę relacji binarnej z poprzednich ćwiczeń. Napisz funktor, który przekształca daną relację w jej domknięcie zwrotno-symetryczne.
- Podaj sygnaturę implementacji struktury danych liczb zespolonych (wyłącznie konstruktory i selektory biegunowe i prostokątne). Sygnatura powinna pasować do dowolnej implementacji: biegunowej, prostokątnej lub mieszanej. Naszkicuj implementację reszty pakietu liczb zespolonych, jako funktor, którego parametrem jest implementacja struktury danych, a wynikiem w pełni funkcjonalny pakiet.
- [SICP] Przypomnijmy sobie pakiet liczb wymiernych z poprzedniego wykładu. Można go zapisać w postaci funktora, sparametryzowanego (wybrany typem) liczb całkowitych. Podaj sygnaturę liczb całkowitych zawierającą wszystkie operacje potrzebne do zaimplementowania pakietu liczb wymiernych ze skracaniem ułamków. (Potrzebne są: operacje arytmetyczne, równość, modulo i ew. porównywanie.) Naszkicuj budowę funktora implementującego taki pakiet liczb wymiernych.
- [SICP] Wyobraźmy sobie pakiet implementujący wielomiany, wraz z różnymi operacjami na wielomianach: suma, różnica, iloczyn, iloraz, reszta z dzielenia, porównanie, pochodna itp. Pakiet taki może być sparametryzowany typem liczb (ciałem), nad którym rozpatrujemy wielomiany. Podaj odpowiednie sygnatury i naszkicuj sposób implementacji pakietu wielomianów jako funktora. Czy implementacja jest na tyle ogólna, aby pakiet działał nie tylko dla liczb zmiennopozycyjnych, ale również dla zespolonych i wymiernych?
- [SICP] Korzystając z wyników poprzednich ćwiczeń podaj, jak można skonstruować pakiet funkcji wymiernych? (Jest to pakiet liczb wymiernych, których współczynniki są wielomianami.)
- [SICP] Korzystając z wyników poprzednich ćwiczeń podaj, jak można skonstruować pakiet wielomianów dwóch zmiennych? (Są to wielomiany jednej zmiennej, których współczynniki są wielomianami drugiej zmiennej.)
- [SICP] Porównaj funktory z mechanizmem dziedziczenia w programowaniu obiektowym. W jaki sposób można zasymulować hierarchię klas za pomocą funktorów? Co z rzutowaniem typów? Co z metodami wirtualnymi? (Nie wiem czy to dobre zadanie. Może tak, a może jest już zbyt późno, ale jakiś związek widzę.)

Uwaga: Zadania oznaczone powyżej [SICP] są mocno inspirowane zadaniami i materiałem z tejże książki, choć nie ma w niej mowy o funktorach.

Wykład 13. Programowanie imperatywne

Paradygmat programowania imperatywnego opiera się na następującej analogii: Zwykle system obliczeniowy modeluje pewien fragment istniejącego świata rzeczywistego. W świecie rzeczywistym mamy do czynienia z obiektami, które w miarę upływu czasu zmieniają swoje stany. W systemie obliczeniowym możemy je modelować za pomocą obiektów obliczeniowych, które też mogą zmieniać stany.

13.1 Referencje

Referencje to najprostsza imperatywna struktura danych. Jest to wskaźnik lub pojemnik, którego zawartość można modyfikować. Intuicyjnie referencja odpowiada adresowi zmiennej w imperatywnych językach programowania, czy tzw. *L*-wartościom⁶. Na referencjach mamy określone następujące trzy operacje:

- referencje, `let r = ref x;;`,
- wyłuskanie wartości, `!r`,
- dynamiczna zmiana wartości, `r := 5`

```
ref : 'a ->'a ref
```

```
!   : 'a ref ->'a
```

```
:=  : 'a ref ->'a ->unit
```

```
⟨wyrażenie⟩ ::=  ref ⟨wyrażenie⟩ |  
                ! ⟨wyrażenie⟩ |  
                ⟨wyrażenie⟩ := ⟨wyrażenie⟩
```

Kłopoty z polimorfizmem. Referencja wskazuje na wartość przynajmniej tak ogólną jak typ referencji. Wyłuskując wskazywaną wartość polimorficzną możemy ją ukonkretnić. Przypisując wartość możemy podać wartość bardziej ogólną. Przypisując wartość konkretniejszą powodujemy ukonkretnienie typu referencji!

```
let f x y = x;;  
val f : 'a ->'b ->'a = <fun>  
let g x y = x + 1;;  
val g : int ->'a ->int = <fun>  
let h x y = x + y;;  
val h : int ->int ->int = <fun>  
let i x y = if y then x else 2 * x;;  
val i : int ->bool ->int = <fun>  
let r = ref g;;  
r := f;;  
r := h;; -- zmienia się typ r!  
r := i;; -- błąd
```

⁶W językach imperatywnych *L*-wartość oznacza nazwę zmiennej lub bardziej skomplikowane wyrażenie, znajdujące się w takim kontekście, w którym reprezentuje adres zmiennej, a nie jej wartość. Na przykład, *L*-wartości występują po lewej stronie przypisania.

13.2 Konstrukcje imperatywne

W momencie, gdy mamy operacje imperatywne, istotne stają się ich efekty uboczne (a niekiedy wartość) oraz kolejność wykonania. Przydatne stają się następujące konstrukcje:

```
⟨wyrażenie⟩ ::= ⟨wyrażenie⟩ { ; ⟨wyrażenie⟩ }+ |  
              begin ⟨wyrażenie⟩ end |  
              if ⟨wyrażenie⟩ then ⟨wyrażenie⟩ |  
              while ⟨wyrażenie⟩ do ⟨wyrażenie⟩ done |  
              for ⟨identyfikator⟩ = ⟨wyrażenie⟩ [ to | downto ] ⟨wyrażenie⟩ do ⟨wyrażenie⟩ done |  
              ...
```

- $w_1; w_2; \dots; w_n$ — sekwencja instrukcji,
- `begin ...end` — nawiasy, blok,
- `if w then x` — jest równoważne wyrażeniu warunkowemu `if w then x else ()`,
- `while w do ... done` — pętla, powtarzamy jej wnętrze dopóki $w = \text{true}$,
- `for v = w_1 to w_2 do ... done` — pętla, v przebiega kolejne wartości całkowite od w_1 do w_2 ,
- wypisywanie:

```
print_int : int ->unit  
  
print_float : float ->unit  
  
print_string : string ->unit  
  
print_newline : unit ->unit
```

- wczytywanie:

```
read_int : unit ->int  
  
read_float : unit ->float  
  
read_line : unit ->string
```

- tablice (poniżej).

13.3 Tablice

Tablice w Ocamlu są jednowymiarowe, indeksowane całkowitymi od 0. Tablice wyżej wymiarowe tworzymy jako tablice tablic. Tablice o elementach typu `t` są typu `t array`. Moduł `Array` zawiera wiele przydatnych operacji na tablicach. Oto najbardziej podstawowe:

- `[|x1; ...; xn|]` — konstruktor tablic,
- `make : int -> 'a -> 'a array`
`make n x` = nowa tablica rozmiaru `n` równa `[|x; ...; x|]`,
- `init : int -> (int -> 'a) -> 'a array`
`init n f` = nowa tablica rozmiaru `n` równa `[|f 0; f 1; ...; f (n-1)|]`,
- `get : 'a array -> int -> 'a`
`a.(n)` = `get a n` = `n`-ty element tablicy `a`, elementy są numerowane od 0 do `length a - 1`,
- `set : 'a array -> int -> 'a -> unit`
`a.(n) <- x = set a n x` przypisanie `n`-temu elementowi tablicy wartości `x`,
- `length : 'a array -> int`
`length a` = długość tablicy,
- `make_matrix : int -> int -> 'a -> 'a array array`
`make_matrix m n x` = macierz, tablica `m`-elementowa tablic `n`-elementowych wypełnionych wartościami `x`,
- `map : ('a -> 'b) -> 'a array -> 'b array`
analogiczne do `List.map`,
- `fold_left : ('a -> 'b -> 'a) -> 'a -> 'b array -> 'a`
analogiczne do `List.fold_left`,
- `fold_right : ('a -> 'b -> 'b) -> 'a array -> 'b -> 'b`
analogiczne do `List.fold_right`.

13.4 Obiekty i stany

W jaki sposób możemy tworzyć obiekty obliczeniowe zdolne do zmiany stanu? Obiektami takimi są referencje. Można jednak tworzyć obiekty lokalne, dostępne tylko wybranym procedurom. Obiekt może być procedurą, wraz z wartościami lokalnymi widocznymi tylko dla niej. Może ona zmieniać te wartości za pomocą przypisania (`:=`).

```
let generator s =
  let r = ref s
  in
  function x ->begin
    r := !r + x;
    !r
  end;;
```

Zauważmy, że każde wywołanie procedury `generator` wiąże się z powstaniem osobnej ramki zawierającej argument `s`. Wynikiem każdego takiego wywołania jest procedura, która może zmieniać wartość zapamiętaną jako `s`.

```
let o = generator 0;;
o 22;;
- : int = 22
o 20;;
- : int = 42
```

13.5 Przykład: konto bankowe

Stan obiektu jest wyznaczony przez zawarte w nim *zmiennie stanowe*. Dobrym zwyczajem jest wprowadzenie bariery abstrakcji oddzielającej zmienne stanowe obiektu od „świata zewnętrznego”. Obiekt powinien natomiast udostępniać metody umożliwiające zmianę jego stanu. Ustalamy więc *interfejs obiektu* złożony z metod zmiany stanu obiektu. Ukrywamy natomiast *implementację* stanu obiektu za pomocą jego zmiennych stanowych.

Możemy tutaj zastosować technikę *przesyłania komunikatów* — to obiekt sam zmienia swój stan na skutek otrzymania komunikatu opisującego zdarzenie powodujące zmianę.

Ponieważ zwykle operujemy na wielu obiektach, powinniśmy zastosować technikę *hermetyzacji*. Każdy obiekt powinniśmy zamknąć w osobnym „pudełku”. Jedyne metody udostępniane przez obiekt mogą mieć dostęp do wnętrza obiektu. Dodatkowo mamy *konstruktor* — procedurę tworzącą nowe obiekty.

```
let konto pocz =
  let saldo = ref pocz
  in
    let wpłata kwota =
      if kwota >0 then begin
        saldo := !saldo + kwota;
        !saldo
      end else failwith "Ujemna lub zerowa kwota"
    and wypłata kwota =
      if kwota >0 then
        if kwota <= !saldo then begin
          saldo := !saldo - kwota;
          !saldo
        end else failwith "Brak środków na koncie"
      else failwith "Ujemna lub zerowa kwota"
    in
      (wpłata,wypłata);;

let (wplac,wyplac) = konto 0;;

wplac 50;;
wyplac 8;;
```

Co by się stało gdyby definicja `konto` nie miała argumentu `pocz`, lecz zaczynała się tak:

```
let konto =  
  let saldo = ref 0  
  in ...
```

13.6 Cena programowania imperatywnego

Konstrukcje imperatywne mają swoją cenę. To programowanie imperatywne wymusza np. wprowadzenie środowiskowego modelu obliczeń w takiej postaci, w jakiej został wcześniej przedstawiony. Ale na tym nie koniec.

Konstrukcje imperatywne powodują kłopoty z pojęciem *tożsamości*. Kiedy x i y to to samo? Czy wtedy, kiedy wartość x i y jest taka sama? Ale przecież wartości obiektów mogą się zmieniać. To może wtedy, gdy każda zmiana wartości x powoduje analogiczną zmianę y ? To nie są łatwe pytania.

Musimy wypracować jakiś język do opisu tożsamości obiektów obliczeniowych. Będziemy mówić, że:

- (w danej chwili) x i y są równe, jeżeli ich wartości są równe, sprawdza to predykat $=$,
- obiekty x i y są tożsame, jeżeli są sobie równe i zmiana wartości jednego powoduje analogiczną zmianę drugiego; tożsamość sprowadza się do tego, że oba obiekty znajdują się w tym samym miejscu w pamięci, sprawdza to predykat $==$,
- obiekty x i y są niezależne, jeżeli zmiana stanu jednego z nich nie powoduje żadnej zmiany stanu drugiego z nich.

Cena imperatywności obejmuje też weryfikację programów imperatywnych i ich pielęgnację. Analiza programów imperatywnych jest bardzo trudna, ze względu na konieczność śledzenia zależności wynikających z kolejności wykonywania instrukcji — „czy ta instrukcja powinna być najpierw, czy tamta?” Ponadto, modyfikowanie programów imperatywnych, dodawanie nowych instrukcji, zmiana kolejności instrukcji, jest źródłem wielu błędów w programach, gdyż łatwo przeoczyć niektóre z zależności. Błędy te nie występują w programach funkcyjnych.

Cena ta rośnie jeszcze bardziej jeżeli dopuścimy *współbieżne* wykonanie kilku procedur operujących na tych samych obiektach. Wówczas bardzo trudno jest przewidzieć wszystkie możliwe scenariusze ich wykonania i przewidzieć wszystkie możliwe przeploty modyfikacji stanów obiektów. Przekonacie się o tym sami na *programowaniu współbieżnym*, che che ...

13.7 Przykład: metoda Monte Carlo

Wynaleziona przez Johna von Neumanna, Stanisława Ulama i Nicholasa Metropolis, w ramach projektu Manhattan. Pierwotnie użyta do badania rozchodzenia się swobodnych neutronów w różnych substancjach. Metoda Monte Carlo to metoda przybliżania probabilistycznego rozmaitych wartości. Polega ona na wielokrotnym losowaniu danych z pewnego zbioru i sprawdzaniu, czy dane te mają określoną własność. W ten sposób przybliżamy prawdopodobieństwo, z jakim dane ze zbioru mają daną własność. Na podstawie przybliżonego prawdopodobieństwa można, z kolei, przybliżyć szukaną wartość.

Badana funkcja na zadanym przedziale musi przyjmować wartości z określonego przedziału. Losujemy punkty na płaszczyźnie, należące do danego przedziału x i y , i badamy z jakim prawdopodobieństwem $y \leq f(x)$. Prawdopodobieństwo tego zdarzenia jest równe stosunkowi

powierzchni badanego fragmentu płaszczyzny leżącego poniżej $f(x)$, do powierzchni całego fragmentu powierzchni.

Spróbujemy zaimplementować metodę Monte Carlo zgodnie z poznanymi zasadami, top-down, rozbijając problemy na podproblemy, stosując bariery abstrakcji, wybierając między programowaniem funkcyjnym i imperatywnym.

Parametry metody to: generator danych, sprawdzana własność i liczba prób. Wynikiem jest przybliżenie prawdopodobieństwa, z jakim dane mają własność.

```
let montecarlo dane wlasnosc liczba_prob =
  let rec montecarlo_rec a n =
    if n = 0 then
      a
    else
      if wlasnosc (dane ()) then
        montecarlo_rec (a+1) (n-1)
      else
        montecarlo_rec a (n-1)
  in
  float (montecarlo_rec 0 liczba_prob) /. float (liczba_prob);;
```

Spróbujmy za pomocą metody Monte Carlo przybliżyć wartość π . Liczba ta jest równa powierzchni koła jednostkowego.

```
let square x = x *. x;;
let kolo (x,y) = square x +. square y <= 1.0;;
let kwadrat () = (losowa_od_do (-1.0) 1.0, losowa_od_do (-1.0) 1.0);;
let pi = 4.0 *. (montecarlo kwadrat kolo 1000);;
```

Pozostał do zaprogramowania generator liczb losowych.

```
let losowa_od_do od doo = losowa () *. (doo -. od) +. od;;

let losowa () =
  let duzo = 1000
  in
  let l = float (losowy_int () mod (duzo + 1)) /. float (duzo)
  in
  if l < 0.0 then (-. l) else l;;

let losowy_int =
  let stan = ref 0
  and a = 937126433
  and b = 937187
  in function () ->
    (
      stan := !stan * b + a;
      !stan
    );;
```

W tym przykładzie użyliśmy niewiele imperatywności, ale miała ona duży wpływ na strukturę programu. W przeciwnym przypadku musielibyśmy przekazywać stan generatora liczb losowych i to w wielu procedurach, przekraczając granice abstrakcji chroniące generator liczb losowych.

13.8 Funkcyjnie, czy imperatywnie?

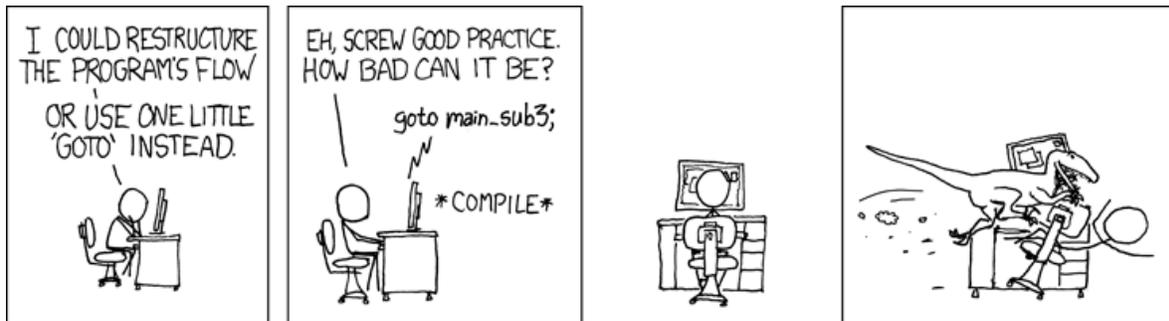
Kiedy programować funkcyjnie, a kiedy imperatywnie? To, że programowanie imperatywne ma swoją cenę, nie oznacza, że wszystko i zawsze należy programować funkcyjnie. Tak jak dobry mechanik powinien znać wszystkie narzędzia i stosować je zgodnie z przeznaczeniem, tak samo dobry programista powinien znać wszystkie techniki programistyczne i paradygmaty programowania i wybierać te, które najlepiej nadają się do rozwiązania danego zadania. Nie wbijamy gwoździ kluczem francuskim i nie dokręcamy ciekącego kranu młotkiem!

Niektóre algorytmy (np. dynamiczne) dużo łatwiej jest zapisać z użyciem *imperatywnych struktur danych*, np. tablic. Nie przeszkadza to jednak takiemu ich zdefiniowaniu, aby cała imperatywność była ukryta przed użytkownikiem, stanowiła sekret implementacji.

Poniżej przedstawiono kolejny przykład mechanizmu, który można zaimplementować elegancko tylko wtedy, gdy jego serce będzie imperatywne.

Niektóre z kolei algorytmy łatwiej jest zapisać funkcyjnie. Dotyczy to zwłaszcza algorytmów operujących na procedurach wyższego rzędu, wszelkiego rodzaju drzewach i algorytmów rekurencyjnych. W takich przypadkach należy stosować programowanie funkcyjne.

13.9 Deser



<http://xkcd.com/292/>

Ćwiczenia

1. Napisz moduł `Counter` o następującej sygnaturze:

```
module type COUNTER = sig
  type counter
  val make : unit ->counter
  val inc : counter ->int
  val reset : unit ->unit
end;;
```

Procedura `make` tworzy nowy licznik o początkowej wartości 0. Procedura `inc` zwiększa licznik o 1 i zwraca jego nową wartość. Procedura `reset` ustawia wartość **wszystkich** liczników na 0.

Podaj złożoność czasową i pamięciową ciągu m operacji, spośród których n to operacje `make`.

2. Dana jest tablica liczb całkowitych zawierająca permutację liczb całkowitych od 0 do n (dla $n \geq 0$). Napisz procedurę `cykl : int array → int`, która wyznacza długość najdłuższego cyklu danej permutacji. Twoja procedura nie może zmieniać danej tablicy. Na przykład:

```
cykl [|2; 1; 0; 5; 6; 4; 3; 8; 7|] = 4
```

3. Dana jest tablica znaków, która wypełniona jest nawiasami. Napisz procedurę `nawiasy : char array → int`, która zwróci maksymalną długość spójnego fragmentu tablicy, który jest poprawnym wyrażeniem nawiasowym.
4. [PCh] Dana jest deklaracja drzew binarnych, w których węzłach znajdują się dodatkowo referencje do węzłów drzewa:

```
type  $\alpha$  tree =
  Null |
  Node of  $\alpha$  *  $\alpha$  tree *  $\alpha$  tree *  $\alpha$  tree ref;;
```

- (a) Drzewo binarne z fastrygą, to drzewo, w którym każdy węzeł posiada dodatkową referencję na następny węzeł w porządku infiksowym. (Ostatni węzeł powinien zawierać referencję na `Null`.) Napisz procedurę `fastryguj : α tree → unit`, która sfastryguje dane drzewo.
- (b) [PCh] Napisz procedurę `lewe_liscie : α tree → unit`, która w każdym węźle drzewa ustawia referencje tak, aby wskazywały na skrajnie lewy liść (`Node`, którego obaj potomkowie, to `Null`) będący potomkiem danego węzła.

- (c) Napisz procedurę `w_lewo : α tree \rightarrow unit`, która w każdym węźle drzewa ustawia referencję tak, aby wskazywały na węzeł (`Node`) znajdujący się na tej samej głębokości w drzewie i najbliższej w lewo od danego węzła. W przypadku skrajnie lewych węzłów, referencja powinna wskazywać na `Null`.
- (d) Napisz procedurę `potomek : α tree \rightarrow unit`, która w każdym węźle ustawi referencję na najgłębiej położony węzeł (`Node`) będący jego potomkiem.
- (e) Powiemy, że drzewo jest *porządkowe* (BST), jeśli w każdym jego wierzchołku postaci `Node(x, left, right, _)` wartości w poddrzewie `left` są mniejsze lub równe `x`, a wartości w poddrzewie `right` są większe równe `x`.
- Napisz procedurę `fastryga : α tree \rightarrow unit`, która mając dane drzewo porządkowe tak ustawi w nim referencje, że węzły zawierające takie same wartości będą połączone w listy cykliczne. W szczególności, jeśli jakaś wartość występuje tylko raz, to referencja w jej węźle powinna być pętelką.

5. Dana jest deklaracja drzew ogólnych:

```
type tree = Node of int * tree list * tree ref
```

Pierwszy argument konstruktora `Node` to wartość przechowywana w węźle, a drugi to lista poddrzew (od lewej do prawej). Liśćmi nazwiemy takie węzły, w których listy poddrzew są puste. Napisz procedurę `fastryga : tree \rightarrow unit`, która dla każdego węzła `u` ustawi referencję (trzeci argument konstruktora `Node`) tak, żeby wskazywała na taki węzeł `v`, który:

- jest potomkiem danego węzła `u` lub `u = v`,
- suma wszystkich liczb w poddrzewie o korzeniu `v` jest maksymalna.

W szczególności, w każdym liściu referencja powinna wskazywać na niego samego.

6. Dana jest tablica par dodatnich liczb rzeczywistych. Liczby te, to długości boków pewnych prostokątów o takich samych polach. Tablica ta jest posortowana rosnąco po pierwszych elementach par, oraz malejąco po drugich elementach par. Napisz procedurę `diagonal : (float * float) array \rightarrow float`, która wyznaczy najkrótszą z przekątnych prostokątów.
7. [Bentley] Dana jest n -elementowa tablica znaków. Napisz procedurę `rotacja : char array \rightarrow int \rightarrow unit`, która rotuje daną tablicę o zadaną liczbę miejsc cyklicznie w prawo, tzn.:

$$\text{rotacja}[[x_1; x_2; \dots; x_n]] \ k$$

zmienia zawartość tablicy na:

$$[[x_{n-k+1}; \dots; x_n; x_1; \dots; x_{n-k}]]$$

Na przykład, wywołanie:

```
rotacja ['a'; 'l'; 'a'; 'm'; 'a'; 'k'; 'o'; 't'; 'a'] 4
```

zmienia zawartość tablicy na `['k'; 'o'; 't'; 'a'; 'a'; 'l'; 'a'; 'm'; 'a']`.

8. Dana jest tablica liczb całkowitych, której pewna rotacja jest posortowana ściśle rosnąco. Napisz procedurę `minmax : int array -> int * int`, która znajduje minimum i maksimum w takiej tablicy.
9. Dana jest niepusta tablica liczb całkowitych $a = [|x_1; \dots; x_n|]$ zawierająca ciąg ściśle monotoniczny, to jest rosnący lub malejący.
Napisz procedurę `blisko_zera : int array -> int`, która zwraca $\min(|x_1|, \dots, |x_n|)$.
10. Tomek chciałby popłynąć z kolegami jachtem w rejs trwający k dni. Potrzebuje zebrać grupę kolegów, którzy chcą i mogą z nim popłynąć. Od każdego ze swoich kolegów dowiedział się od kiedy do kiedy (włącznie) dysponuje wolnym czasem. Teraz zastanawia się, kiedy wyruszyć w rejs, tak żeby mogli żeglować jak największą grupą. W trakcie rejsu koledzy Tomka mogą się zmieniać (o ile obie osoby mają tego samego dnia wolne).
Napisz procedurę `rejs : int -> (int * int) list -> int`, która mając daną liczbę k oraz informacje o dostępności kolegów Tomka zwróci maksymalną wielkość załogi (nie licząc Tomka). Jeżeli rejsu nie da się zorganizować, to poprawnym wynikiem jest 0. Na przykład: `rejs 20 [(12, 36); (48, 100); (28, 70); (55, 80); (0, 65); (25, 30)] = 3`.
11. Tomek chciałby popłynąć z kolegami jachtem w rejs. Potrzebuje zebrać grupę k kolegów, którzy chcą i mogą z nim popłynąć. Od każdego ze swoich kolegów dowiedział się od kiedy do kiedy (włącznie) dysponuje wolnym czasem. Teraz zastanawia się, kiedy wyruszyć w rejs, tak żeby mógł on trwać jak najdłużej. W trakcie rejsu załoga nie może się zmieniać.
Napisz procedurę `rejs : int -> (int * int) list -> int`, która mając daną liczbę k oraz informacje o dostępności kolegów Tomka zwróci maksymalną długość rejsu. Jeżeli rejsu nie da się zorganizować, to poprawnym wynikiem jest 0.
Na przykład: `rejs 2 [(12, 36); (48, 100); (28, 70); (50, 80); (0, 69); (25, 30)] = 42`.
12. [PCH] Dana jest tablica liczb całkowitych $a = [|x_1; x_2; \dots; x_n|]$. Tablica ta ma taką własność, że liczby $x_{i+1} - x_i$, dla $i = 1, 2, \dots, n - 1$, tworzą ciąg rosnący.
- Napisz procedurę `rozne : int array -> bool`, która dla danej tablicy sprawdza, czy jej elementy są parami różne.
 - Napisz procedurę `minimum : int array -> int`, która dla danej tablicy wyznaczy minimum z jej elementów.
- Rozwiązując to zadanie nie wolno Ci tworzyć żadnych własnych procedur rekurencyjnych. Wolno natomiast korzystać z pętli i innych konstrukcji imperatywnych.
Podaj złożoność czasową i pamięciową swojego rozwiązania.
13. Dana jest tablica liczb całkowitych $a = [|x_1; x_2; \dots; x_n|]$. Napisz procedurę `segment : int array -> int`, która wyznaczy długość $(j - i + 1)$ najdłuższego takiego spójnego fragmentu tablicy a , $[|x_i; x_{i+1}; \dots; x_j|]$, dla którego $\min(x_i, x_{i+1}, \dots, x_j) \geq j - i + 1$.
Rozwiązując to zadanie nie wolno Ci tworzyć żadnych własnych procedur rekurencyjnych. Zamiast tego powinieneś użyć konstrukcji imperatywnych.
14. Dana jest tablica liczb całkowitych $a = [|x_1; x_2; \dots; x_n|]$. Napisz procedurę `daleko : int array -> (int * int)`, która wyznaczy taką parę indeksów (i, j) , że:

- $i < j$,
- $a.(i) < a.(j)$,
- $j - i$ jest maksymalne.

Rozwiązując to zadanie nie wolno Ci tworzyć żadnych własnych procedur rekurencyjnych. Zamiast tego powinieneś użyć konstrukcji imperatywnych.

15. [PCh] Dane są dwie listy (bez zapętleń), które łączą się i od pewnego miejsca są tą samą listą. Znajdź ten wspólny fragment.
16. Napisz procedurę sprawdzającą, czy dana lista zawiera cykl. Czy potrafisz wyznaczyć jego długość?

Wytyczne dla prowadzących ćwiczenia

- Ad. 1** Wszystkie operacje powinny działać w czasie stałym. Należy „stempla” z momentem ostatniego resetowania.
- Ad. 4a** Należy użyć metody przechadzki.
- Ad. 4b** To zadanie można rozwiązać w stałej pamięci (choć niekoniecznie jest to najładniejsze rozwiązanie). Za pomocą nie ustawionych jeszcze referencji można zasymulować stos do przeszukiwania drzewa włąb.
- Ad. 4c** To zadanie można rozwiązać w stałej pamięci (choć niekoniecznie jest to najładniejsze rozwiązanie). Za pomocą nie ustawionych jeszcze referencji można zasymulować kolejkę do przeszukiwania drzewa wszere.
- Ad. 16** Można to zrobić w stałej pamięci i liniowym czasie (i to na kilka sposobów)!

Wykład 14. Imperatywne wskaźnikowe struktury danych

W poprzednim wykładzie przedstawione zostały imperatywne mechanizmy w języku programowania. W tym wykładzie zobaczymy na przykładach, jak mechanizmy te mogą zostać wykorzystane do implementacji efektywnych struktur danych.

14.1 Dwustronne kolejki ze scalaniem i odwracaniem

Przypuśćmy, że potrzebujemy zaimplementować dwustronne kolejki, z dodatkowymi operacjami: scalaniem i odwracaniem. Scalenie dwóch kolejek powoduje, że pierwsza z nich staje się wynikiem ich sklejenia, a druga staje się pusta. Odwracanie powoduje odwrócenie kierunku kolejki. Operacje te ujmując następująca sygnatura:

```
module type QUEUE =
  sig
    type  $\alpha$  queue
    exception EmptyQueue
    val init : unit  $\rightarrow$   $\alpha$  queue
    val is_empty :  $\alpha$  queue  $\rightarrow$  bool
    val put_first :  $\alpha$  queue  $\rightarrow$   $\alpha$   $\rightarrow$  unit
    val put_last :  $\alpha$  queue  $\rightarrow$   $\alpha$   $\rightarrow$  unit
    val first :  $\alpha$  queue  $\rightarrow$   $\alpha$ 
    val last :  $\alpha$  queue  $\rightarrow$   $\alpha$ 
    val remove_first :  $\alpha$  queue  $\rightarrow$  unit
    val remove_last :  $\alpha$  queue  $\rightarrow$  unit
    val merge :  $\alpha$  queue  $\rightarrow$   $\alpha$  queue  $\rightarrow$  unit
    val rev :  $\alpha$  queue  $\rightarrow$  unit
  end;;
```

14.1.1 Implementacja

- Technika atrap i strażników.
- Lista dwukierunkowa, bez określania kierunków dowiązań. Atrapy na końcach. Lista, to para dowiązań do atrap.

```
type  $\alpha$  elem = { mutable l1 :  $\alpha$  elem; mutable l2 :  $\alpha$  elem; v :  $\alpha$  option }
type  $\alpha$  queue = { mutable front :  $\alpha$  elem; mutable back :  $\alpha$  elem }
exception EmptyQueue
```

- Niezmiennik struktury danych, oraz pomocnicze warunki — przydatne przy testowaniu i w asercjach:
 - Warunek testowy, że dwa elementy listy są sąsiednie:

```
let test_neighbours e1 e2 =
  (e1.l1 == e2 || e1.l2 == e2) &&
  (e2.l1 == e1 || e2.l2 == e1) &&
  (e1 != e2)
```

– Warunek testowy atrapy na końcu listy:

```
let test_decoy_elem e =  
  e.l1 == e && e.v = None
```

– Warunek testowy elementu wewnątrz listy:

```
let test_inside_elem e =  
  test_neighbours e e.l1 &&  
  test_neighbours e e.l2 &&  
  e.l1 != e.l2 &&  
  is_some e.v
```

– Niezmiennik struktury danych:

```
let test_invariant q =  
  let forall q f =  
    let rec iter_elem e f prev acc =  
      if e.v <>None then  
        let next = if prev == e.l1 then e.l2 else e.l1  
        in iter_elem next f e (acc && f e)  
      else acc  
    in iter_elem q.front.l2 f q.front true  
  in  
    test_decoy_elem q.front &&  
    test_decoy_elem q.back &&  
    forall q test_inside_elem
```

• Operacje:

(* Wartość elementu. *)

```
let value e =  
  match e.v with  
  Some x ->x |  
  None ->raise EmptyQueue
```

(* Predykat sprawdzający czy kolejka jest pusta. *)

```
let is_empty q =  
  q.front.l2 == q.back
```

(* Konstruktor pustej kolejki. *)

```
let init () =  
  let rec g1 = { l1 = g1; l2 = g2; v = None}
```

```

and g2 = { l1 = g2; l2 = g1; v = None}
in
  let res = { front = g1; back = g2 }
  in begin
    assert (test_invariant res && test_neighbours g1 g2);
    res
  end

(* Wstawia nowy element z wartością x pomiędzy elementy p i q. *)
let put_between p q x =
  let r = { l1 = p; l2 = q; v = Some x }
  in begin
    assert (test_neighbours p q);
    if p.l1 == q then p.l1 <- r else p.l2 <- r;
    if q.l1 == p then q.l1 <- r else q.l2 <- r;
    assert (test_neighbours p r && test_neighbours q r);
  end

(* Wstawienie na początek. *)
let put_first q x =
  put_between q.front q.front.l2 x

(* Wstawienie na koniec. *)
let put_last q x =
  put_between q.back q.back.l2 x

(* Pierwszy element. *)
let first q =
  value q.front.l2

(* Ostatni element. *)
let last q =
  value q.back.l2

(* Usuwa element (nie będący atrapą). *)
let remove e =
  assert (test_inside_elem e);
  let e1 = e.l1
  and e2 = e.l2

```

```

in begin
  if e1.l1 == e then e1.l1 <- e2 else e1.l2 <- e2;
  if e2.l1 == e then e2.l1 <- e1 else e2.l2 <- e1;
  assert (test_neighbours e1 e2)
end

(* Usunięcie pierwszego el. *)
let remove_first q =
  if is_empty q then
    raise EmptyQueue
  else
    remove q.front.l2

(* Usunięcie ostatniego el. *)
let remove_last q =
  if is_empty q then
    raise EmptyQueue
  else
    remove q.back.l2

(* Sklejenie dwóch kolejek, druga kolejka staje się pusta. *)
let merge q1 q2 =
  assert (not (q1 == q2));
  let f1 = q1.front
  and b1 = q1.back
  and f2 = q2.front
  and b2 = q2.back
  in
    assert (not (f1==b1) && not (f2==b2));
    let e1 = b1.l2
    and e2 = f2.l2
    in begin
      (* Złączenie kolejek. *)
      if e1.l1 == b1 then e1.l1 <- e2 else e1.l2 <- e2;
      if e2.l1 == f2 then e2.l1 <- e1 else e2.l2 <- e1;
      (* Wynik w q1 *)
      q1.back <- b2;
      (* q2 puste *)
      f2.l2 <- b1;

```

```

        b1.l2 <- f2;
        q2.back <- b1
    end

    (* Odwrócenie kolejki. *)
let rev q =
    let f = q.front
    and b = q.back
    in begin
        q.front <- b;
        q.back <- f
    end

    (* Testy *)
    :

```

W oczywisty sposób, złożoność wszystkich operacji na kolejkach jest rzędu $O(1)$.

14.2 Drzewa Find-Union

Wyobraźmy sobie, że potrzebujemy struktury danych do reprezentowania relacji równoważności oraz jej klas abstrakcji. Potrzebne są następujące typy i operacje:

- `type α set` — typ reprezentujący klasy abstrakcji elementów typu α ,
- `make_set : $\alpha \rightarrow \alpha$ set` — tworzy jednoelementową klasę abstrakcji zawierającą x ,
- `find : α set $\rightarrow \alpha$` — zwraca ustalonego reprezentanta klasy abstrakcji,
- `equivalent : α set $\rightarrow \alpha$ set \rightarrow bool` — sprawdza, czy dwie klasy abstrakcji są tą samą klasą abstrakcji,
- `union : α set $\rightarrow \alpha$ set \rightarrow unit` — skleja dwie klasy abstrakcji, tworząc nową klasę abstrakcji, której reprezentantem jest reprezentant jednej ze sklejaných klas,
- `n_of_sets : unit \rightarrow int` — zwraca liczbę rozłącznych klas abstrakcji,
- `elements : α set $\rightarrow \alpha$ list` — zwraca listę elementów tworzących klasę abstrakcji.

Powyższy typ i operacje można ująć w następującą sygnaturę:

```

module type FIND_UNION = sig
    type 'a set
    val make_set : 'a ->'a set
    val find : 'a set ->'a
    val equivalent : 'a set ->'a set ->bool
    val union : 'a set ->'a set ->unit

```

```

    val elements : 'a set ->'a list
    val n_of_sets : unit->int
end;;

```

Wartości typu α `set` reprezentują klasy abstrakcji zawierające wartości typu α , jednak dopuszczamy, aby różne (w sensie `==`) wartości typu α `set` reprezentowały tę samą klasę abstrakcji.

Każde wywołanie `make_set` tworzy odrębną, jednoelementową klasę abstrakcji. Jeżeli wszystkie wartości, na których wykonano operację `make_set` są różne, to można przyjąć, że dwa elementy x i y są w tej samej klasie abstrakcji wtw., gdy dla wygenerowanych dla nich klas abstrakcji c_x i c_y zachodzi `find c_x = find c_y`. Jeśli tak nie jest, to argumenty `make_set` i wyniki `find` należy traktować jak etykiety właściwych elementów klas abstrakcji, które nie muszą być unikalne.

Zauważmy, że implementacja tak wyspecyfikowanej struktury danych **musi być imperatywna**. Wykonanie operacji `union` wpływa nie tylko na jej argumenty, ale również na wszystkie wartości (wygenerowane jako wyniki `make_set`) reprezentujące scalane klasy abstrakcji. Dodatkowo, efektywna implementacja wymaga zastosowania wskaźnikowej struktury danych.

Przykład: Efekty uboczne operacji `make_set`:

```

n_of_sets();;
- : int = 0
let a = make_set "a"
and b = make_set "b"
and c = make_set "c"
and d = make_set "d";;
n_of_sets();;
- : int = 4
union a b;;
union c d;;
n_of_sets();;
- : int = 2
a == d;;
- : bool = false
a == b;;
- : bool = false
equivalent a b;;
- : bool = true
equivalent a d;;
- : bool = false
union b c;;
equivalent a d;;
- : bool = true

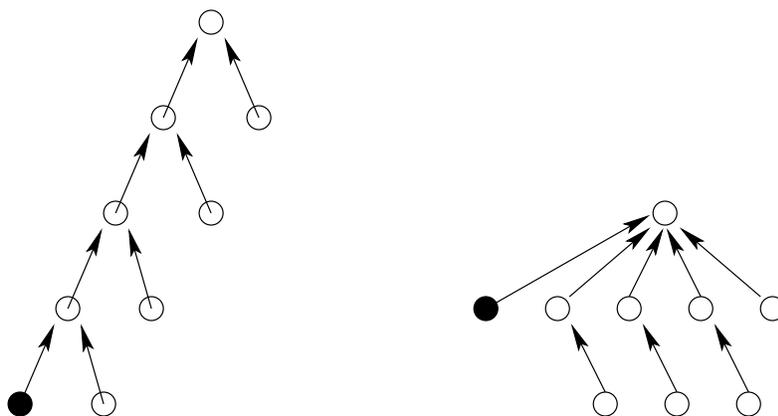
```

14.2.1 Implementacja

Klasy abstrakcji będziemy implementować w postaci drzew, których węzły reprezentują elementy klas abstrakcji. Przy tym z każdego węzła będzie wychodzić wskaźnik do jego ojca.

Reprezentantem klasy będzie korzeń drzewa. Wskaźnik wychodzący z korzenia prowadzi do niego samego. Wynikiem operacji `make_set` jest utworzenie wierzchołka, z którego wychodzi wskaźnik wskazujący na niego samego. Operacja `find` polega na przejściu wzdłuż wskaźników aż do korzenia drzewa i zwróceniu reprezentowanego przez niego elementu. Operacja `union` polega na znalezieniu korzeni dwóch drzew i podłączeniu jednego jako syna drugiego. Dodatkowo stosujemy dwie optymalizacje.

Kompresja ścieżek Za każdym razem, gdy wyszukujemy reprezentanta klasy, we wszystkich węzłach na ścieżce od danego węzła do korzenia przestawiamy wskaźniki na korzeń drzewa.



Rysunek 2: Kompresja ścieżek

Wybór nowego korzenia W momencie, gdy scalamy dwa drzewa, możemy wybrać korzeń, którego z nich będzie korzeniem nowego drzewa. Generalnie, powinniśmy podłączać mniejsze drzewo do większego. Nie będziemy jednak pamiętać ani wielkości poddrzew zakorzenionych w poszczególnych wierzchołkach, ani ich wysokości. Zamiast tego, każdemu wierzchołkowi przypiszemy *rangę*, liczbę całkowitą, która jest górnym ograniczeniem na wysokość danego wierzchołka. W momencie tworzenia wierzchołka, jego ranga wynosi 0. Kompresja ścieżek nie wpływa na rangi wierzchołków. Gdy scalamy dwa drzewa, to podłączamy korzeń o mniejszej randze, do korzenia o większej randze. Jeżeli oba korzenie mają tę samą rangę, to wybieramy którykolwiek z nich i zwiększamy jego rangę o 1.

Wyliczanie elementów klas i licznik klas Opisana struktura danych pozwala na szybką implementację operacji `find` i `union`, ale nie pozwala na wyliczanie elementów klas. Potrzebujemy do tego wskaźników idących w dół drzewa. Dla każdej klasy abstrakcji utrzymujemy również drzewo „rozpinające” elementy klasy. Korzeniem tego drzewa jest reprezentant klasy abstrakcji. W każdym węzle drzewa pamiętamy listę jego następników. Kształty drzew tworzonych przez wskaźniki idące „w górę” i „w dół” nie muszą się pokrywać.

Dodatkowo, aby odpowiadać na pytania o liczbę klas abstrakcji, utrzymujemy licznik klas.

```
module Find_Union : FIND_UNION = struct
  (* Typ klas elementów typu 'a. *)
```

```

type 'a set = {
  elem      : 'a;          (* Element klasy. *)
  up        : 'a set ref;  (* Przodek w drzewie find-union. *)
  mutable rank : int;      (* Ranga w drzewie find-union. *)
  mutable next : 'a set list (* Lista potomków w pewnym drzewie rozpinającym
klasę. *)
}

(* Licznik klas. *)
let sets_counter = ref 0

(* Liczba wszystkich klas. *)
let n_of_sets () = !sets_counter

(* Tworzy nową klasę złożoną tylko z danego elementu. *)
let make_set x =
  let rec v = { elem = x; up = ref v; rank = 0; next = [] }
  in begin
    sets_counter := !sets_counter + 1;
    v
  end

(* Znajduje korzeń drzewa, kompresując ścieżkę. *)
let rec go_up s =
  if s == !(s.up) then s
  else begin
    s.up := go_up !(s.up);
    !(s.up)
  end

(* Znajduje reprezentanta danej klasy. *)
let find s =
  (go_up s).elem

(* Sprawdza, czy dwa elementy są równoważne. *)
let equivalent s1 s2 =
  go_up s1 == go_up s2

(* Scala dwie dane (rozłączne) klasy. *)
let union x y =
  let fx = go_up x
  and fy = go_up y
  in
  if not (fx == fy) then begin
    if fy.rank > fx.rank then begin
      fx.up := fy;
      fy.next <- fx :: fy.next
    end else begin
      fy.up := fx;
      fx.next <- fy :: fx.next;
      if fx.rank = fy.rank then fx.rank <- fy.rank + 1
    end;
    sets_counter := !sets_counter - 1
  end

```

```

end

(* Lista elementów klasy. *)
let elements s =
  let acc = ref []
  in
    let rec traverse s1 =
      begin
        acc := s1.elem :: !acc;
        List.iter traverse s1.next
      end
    in begin
      traverse (go_up s);
      !acc
    end
end

end;;

```

14.2.2 Analiza kosztu

W niniejszym punkcie zajmiemy się analizą łącznego kosztu wykonania m operacji, spośród których n to operacje `make_set`. Na początek przeanalizujemy kilka własności rang.

Lemat 3. *Dla każdego węzła, który nie jest korzeniem, jego ranga jest mniejsza od rangi jego ojca.*

Dowód. Dowód przebiega indukcyjnie po historii obliczeń. Operacja `union` podłączając jeden korzeń do drugiego zapewnia, że ojciec ma większą rangę niż nowy syn. Kompresja ścieżek również zachowuje powyższą własność. \square

Lemat 4. *Dla dowolnego drzewa, którego korzeń ma rangę r , liczba węzłów w drzewie wynosi przynajmniej 2^r .*

Dowód. Dowód przebiega indukcyjnie po historii obliczeń. Oczywiście jednoelementowe drzewa tworzone przez `make_set` spełniają lemat. Kompresja ścieżek nie zmienia liczby węzłów w drzewie, ani rangi korzenia. Jeżeli scalamy drzewa, których korzenie mają różne rangi, to oczywiście lemat jest zachowany. Jeżeli scalamy drzewa, których korzenie mają taką samą rangę r , to każde z tych drzew ma przynajmniej 2^r węzłów, a po ich scaleniu powstaje drzewo, którego korzeń ma rangę $r + 1$ i które zawiera przynajmniej 2^{r+1} elementów. \square

Fakt 4. *Dla dowolnej rangi r istnieje co najwyżej $\frac{n}{2^r}$ węzłów rangi r lub większej.*

Dowód. Ustalmy rangę r . Ranga wierzchołków rośnie w wyniku operacji `union`. Usuńmy z naszego ciągu instrukcji wszystkie operacje `union`, które prowadzą do powstania wierzchołków o randze większej niż r . W rezultacie każdy wierzchołek, który oryginalnie miał rangę równą r lub większą, ma rangę równą r i jest korzeniem pewnego drzewa.

Z lematu 4 wynika, że każdy taki wierzchołek jest korzeniem drzewa zawierającego przynajmniej 2^r węzłów. Ponieważ różne drzewa są rozłączne, a wszystkich wierzchołków jest n , więc wierzchołków rangi r (lub większej) może być co najwyżej $\frac{n}{2^r}$. \square

Fakt 5. *Rangi węzłów nie przekraczają $\lfloor \log_2 n \rfloor$.*

Dowód. Jest to natychmiastowy wniosek z faktu 4. □

Def. 1. Funkcja $\log_2^i x$ jest zdefiniowana standardowo, przy czym może ona być nieokreślona. Funkcja \log^* jest zdefiniowana następująco:

$$\log^* x = \min\{i \geq 0 : \log_2^i x \leq 1\}$$

□

Przykładowo, $\log^* 4 = 2$, $\log^* 65000 = 4$.

Tw. 4. Łączny koszt wykonania m operacji na drzewach *find-union* (bez operacji *elements*), spośród których n to operacje *make_set* wynosi $O(m \cdot \log^* n)$.

Dowód. Zdefiniujemy pomocniczą funkcję B , odwrotną do funkcji \log^* :

$$B(j) = \begin{cases} \left. \begin{matrix} 2^{\cdot^{\cdot^{\cdot^2}}} \\ 2 \end{matrix} \right\} j \text{ razy} & \text{jeśli } j \geq 1 \\ 1 & \text{jeśli } j = 0 \\ -1 & \text{jeśli } j = -1 \end{cases}$$

Wierzchołki dzielimy na *bloki*, ze względu na \log^* od ich rang. Bloki numerujemy tymi wartościami. Ponieważ bloki mają numery od 0 do co najwyżej $\log^*(\log_2 n) = \log^* n - 1$, więc wszystkich bloków jest co najwyżej $\log^* n$. Blok numer j zawiera rangi od $B(j-1) + 1$ do $B(j)$.

Jeżeli pominiemy koszt operacji *go_up*, to pozostały koszt operacji wynosi $O(m)$. Możemy więc skupić się na koszcie operacji *go_up*. Pojedyncza operacja *go_up* przebiega ścieżkę od danego wężła do syna korzenia: (x_0, x_1, \dots, x_l) . Rangi wierzchołków na ścieżce rosną. Wierzchołki na ścieżce należące do tego samego bloku występują po kolei.

Koszt operacji *go_up* przypisujemy wierzchołkom na ścieżce, przy czym dzielimy go na koszt *związany ze ścieżką* i koszt *związany z blokami*. Koszt związany z blokami przypisujemy ostatnim wierzchołkom na ścieżce należącym do poszczególnych bloków, oraz synowi korzenia, x_{l-1} . Koszt związany z blokami każdej operacji *go_up* wynosi co najwyżej jeden plus liczba bloków, czyli $O(1 + \log^* n)$. Tak więc koszt związany z blokami wszystkich operacji wynosi $O(m \cdot \log^* n)$. Należy jeszcze pokazać, że łączny koszt związany ze ścieżkami jest również rzędu $O(m \cdot \log^* n)$.

Zauważmy, że jeżeli jakiemuś wierzchołkowi został raz przydzielony koszt za blok, to już więcej nie będzie mu przydzielony koszt za ścieżkę. Jeżeli natomiast wierzchołkowi został w danej operacji *go_up* przydzielony koszt za ścieżkę, to w wyniku tej operacji zmienił się ojciec tego wierzchołka i to na taki o wyższej randze. Jeżeli ten nowy ojciec należy do innego bloku, to dany wierzchołek już więcej nie będzie miał przypisanego kosztu za ścieżkę. Tak więc, jeżeli dany wierzchołek należy do bloku nr j , to może on mieć przypisany koszt za ścieżkę co najwyżej $B(j) - B(j-1) - 1$ razy.

Oznaczmy przez $N(j)$ liczbę wierzchołków należących do bloku j . Korzystając z faktu, że węzłów o randze $\geq r$ jest $\leq \frac{n}{2^r}$ mamy:

$$\begin{aligned} N(0) &\leq \frac{n}{2^0} = \frac{n}{B(0)} \\ N(j) &\leq \frac{n}{2^{B(j-1)+1}} = \frac{n}{2B(j)} < \frac{n}{B(j)} \end{aligned}$$

Łączną liczbę kosztów za ścieżki możemy oszacować mnożąc, dla każdego bloku, maksymalną liczbę wierzchołków w tym bloku przez maksymalną liczbę opłat za ścieżkę.

$$\begin{aligned} \sum_{j=0}^{\log^* n-1} (N(j) \cdot (B(j) - B(j-1) - 1)) &\leq \sum_{j=0}^{\log^* n-1} \left(\frac{n}{B(j)} \cdot (B(j) - B(j-1) - 1) \right) \leq \\ &\leq \sum_{j=0}^{\log^* n-1} \left(\frac{n}{B(j)} \cdot B(j) \right) = \sum_{j=0}^{\log^* n-1} (n) \leq n \cdot \log^* n \end{aligned}$$

Tak więc łączny koszt wszystkich operacji jest rzędu $O(m \log^* n)$. □

W praktyce czynnik $\log^* n$ można przyjąć za stały, gdyż nie przekracza on 5 dla wszelkich praktycznych wielkości. Zauważmy, że $B(5) \geq 10^{19728}$, natomiast liczba wszystkich atomów we Wszechświecie jest szacowana na 10^{80} .

Powyższe twierdzenie można rozszerzyć również na operacje `elements`, jeżeli np. zapewnimy, że każdy element pojawia się na listach elementów klas tylko raz. Wówczas łączny czas wykonania tych operacji nie przekracza n .

Funkcja Ackermanna (zgodnie z jej sformułowaniem podanym przez Hermesa [Her65, Bra83]) jest zdefiniowana następująco:

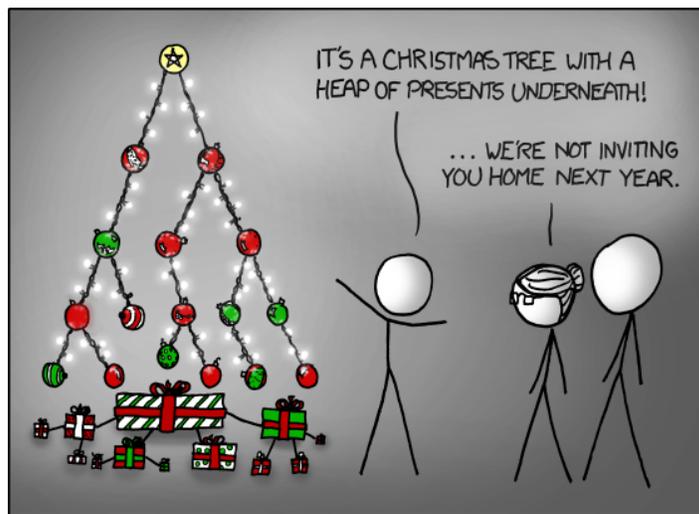
$$\begin{aligned} A(0, i) &= i + 1 \\ A(i, 0) &= A(i - 1, 1) \\ A(i, j) &= A(i - 1, A(i, j - 1)) \end{aligned}$$

Można pokazać, że łączny koszt m operacji, spośród których n to operacje `make_set`, jest jeszcze mniejszy, mianowicie jest rzędu $O(m \cdot \alpha(m, n))$, gdzie $\alpha(m, n)$ jest funkcją „odwrotną” do funkcji Ackermanna:

$$\alpha(m, n) = \min \left\{ i \geq 1 : A \left(i, \left\lfloor \frac{m}{n} \right\rfloor \right) > \log_2 n \right\}$$

W praktyce czynnik $\alpha(m, n)$ można przyjąć za stały, gdyż nie przekracza on 4 dla wszelkich praktycznych wielkości.

14.3 Deser



<http://xkcd.com/835/>

Ćwiczenia

1. Zaimplementuj kolejki FIFO+LIFO jako imperatywne listy dwukierunkowe.
2. Dana jest definicja typu elementów tworzących listy wskaźnikowe:

```
type  $\alpha$  option = None | Some of  $\alpha$ 
type  $\alpha$  elem = {v:  $\alpha$ ; mutable next:  $\alpha$  lista}
and  $\alpha$  lista =  $\alpha$  elem option
```

- (a) Napisz procedurę `petla : lista → unit`, która mając daną listę jednokierunkową, tworzy z niej listę cykliczną, ale z odwróconym kierunkiem wskaźników. Możesz założyć, że dana lista jest poprawną listą jednokierunkową, to znaczy ma koniec i nie zapętla się.
 - (b) Napisz procedurę `przeplot : lista → lista → unit`, która splata obie listy w jedną listę postaci: pierwszy rekord pierwszej listy, pierwszy rekord drugiej listy, drugi rekord pierwszej listy, drugi rekord drugiej listy, itd. Jeśli jedna z list jest dłuższa, to jej końcówka stanowi końcówkę listy wynikowej.
3. Napisz procedurę tworzącą cykliczną listę modyfikowalną.
 4. Dane są deklaracje typów:

```
type elem = { x:int; mutable prev:elem list };;
type lista = elem list;;
```

Napisz procedurę `ustaw : lista → unit`, która dla danej listy $[x_1; x_2; \dots; x_n]$ typu `lista`, tak ustawia pola `prev`, aby (dla $i = 1, 2, \dots, n$) prowadziło ono z elementu x_i do listy zaczynającej się w elemencie x_{n-i+1} .

5. [XII OI, Skarbonki] Mamy n skarbonek otwieranych kluczykami. Do każdej skarbonki jest inny klucz. Kluczyki do skarbonek są powrzućane do skarbonek. Żeby dostać się do zawartości skarbonki, skarbonek można otworzyć kluczykiem (jeśli się go ma) lub ją rozbić.

Skarbonki są ponumerowane od 0 do $n - 1$. Na podstawie tablicy a (rozmiaru n), takiej, że $a[i] =$ numer skarbonki, w której znajduje się klucz do skarbonki numer i , oblicz minimalną liczbę skarbonek, które należy rozbić, tak aby dostać się do zawartości wszystkich skarbonek.

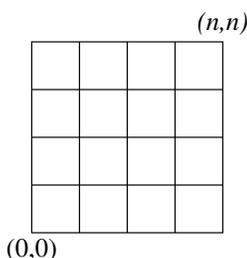
6. [X OI, Małpki]

[[A może by to przenieść do wykładu jako przykład zastosowania find-union?]]

Na drzewie wisi n małpek ponumerowanych od 1 do n . Małpka z nr 1 trzyma się gałęzi ogonkiem. Pozostałe małpki albo są trzymane przez inne małpki, albo trzymają się innych małpek, albo jedno i drugie równocześnie. Każda małpka ma dwie przednie łapki, każdą może trzymać co najwyżej jedną inną małpkę (za ogon). Rozpoczynając od chwili 0, co sekundę jedna z małpek puszcza jedną łapkę. W ten sposób niektóre małpki spadają na ziemię, gdzie dalej mogą puszczać łapki (czas spadania małpek jest pomijalnie mały).

Zaprojektuj algorytm, który na podstawie opisu tego która małpka trzyma którą, oraz na podstawie opisu łapek puszcanych w kolejnych chwilach, dla każdej małpki wyznaczy moment, kiedy spadnie ona na ziemię.

- Na szachownicy jest ustawionych n wież, które należy pokolorować. Jeśli dwie wieże się atakują (są w tej samej kolumnie lub wierszu), to muszą być tego samego koloru. Napisz procedurę `kolory : int * int list → int`, która na podstawie listy współrzędnych wież wyznaczy maksymalną liczbę kolorów, których można użyć kolorując wieże. Podaj złożoność czasową i pamięciową swojego rozwiązania.
- Mamy daną kratownicę złożoną z przewodów elektrycznych wielkości $n \times n$.



Niestety, poszczególne przewody przepalają się. Dana jest lista kolejnych jednostkowych przewodów, które się przepalają. Każdy przewód jest opisany trójką (x, y, v) , gdzie (x, y) to współrzędna jego lewego/dolnego końca, a v jest wartością logiczną równą `true` dla przewodów pionowych i `false` dla przewodów poziomych.

Napisz procedurę `przewody : int -> (int * int * bool) list -> int`, która dla danej liczby n i listy kolejnych przewodów przepalających się, określi po przepaleniu ilu przewodów prąd nie może płynąć między punktami $(0,0)$ i (n,n) . Jeżeli po przepaleniu się wszystkich przewodów z listy prąd nadal może płynąć, poprawnym wynikiem jest -1 .

- Dana jest prostokątna mapa złóż gazu wraz z odwiertami pobierającymi gaz. Mapa jest dana w postaci dwuwymiarowej prostokątnej tablicy liczb całkowitych. Miejsca odwiertów są zaznaczone liczbami ujemnymi. Wartość bezwzględna elementu tablicy oznacza ilość znajdującego się w danym miejscu gazu.

Pola niezerowe stykające się bokami tworzą jedno złożo. Odwierty znajdujące się w danym złożu pozwalają pobrać tyle gazu ile pól obejmuje złożo, po czym ilość gazu we wszystkich polach złoża maleje o 1. Może to spowodować zmniejszenie lub podzielenie się złoża. Dalej odwierty pozwalają pobierać gaz z tak zmienionych złóż.

Napisz procedurę `gaz : int array array → int`, która obliczy łączną ilość gazu, jaką wydobędą odwierty.

- W akademiku mieszka n informatyków, a każdy z nich ma komputer. Niektórzy z informatyków połączyli swoje komputery za pomocą sieci lokalnych. Komputery należące do tej samej sieci lokalnej mogą się ze sobą komunikować. Niektóre komputery należą do więcej niż jednej sieci lokalnej.

Informatycy postanowili połączyć lokalne sieci w jedną inter-sieć obejmującą wszystkie komputery. Okazało się jednak, że między niektórymi komputerami nie ma którejdy

przesyłać informacji. Należy stworzyć jeszcze jedną sieć lokalną, realizującą brakujące połączenia. Ta dodatkowa sieć powinna obejmować jak najmniejszą liczbę komputerów.

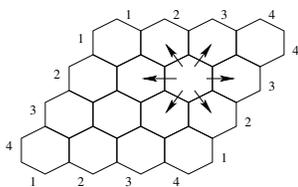
Komputery są ponumerowane od 1 do n . Napisz procedurę `sieć : int → int list list → int`, która na podstawie liczby komputerów n oraz listy list komputerów należących do poszczególnych sieci lokalnych, wyznaczy ile komputerów należy połączyć dodatkową siecią lokalną.

Na przykład, dla danych: `sieć 8 [[1;3;2]; [7;2]; [1;8]; [5;6]]` poprawnym wynikiem jest 3; nowa sieć może obejmować np. komputery 3, 4 i 5 lub 4, 6 i 7.

11. Dana jest plansza podzielona na sześciokątne pola, o wymiarach $n \times n$ (patrz rysunek). i -

te pole w j -tym rzędzie sąsiaduje z polami:

- $i - 1$ i i w rzędzie $j - 1$,
- $i - 1$ i $i + 1$ w rzędzie j ,
- i i $i + 1$ w rzędzie $j + 1$.



Dwaj gracze, biały i czarny, grają w grę *hex*. Zajmują oni na

przemian wolne pola na planszy. Zaczyna gracz biały. Gracz biały stara się połączyć swoimi polami lewy i prawy brzeg planszy, a gracz czarny górny i dolny brzeg planszy. Wygrywa gracz, który jako pierwszy połączy odpowiednie brzegi planszy, w ten sposób, że z jednego brzegu na drugi będzie można przejść przez sąsiadujące pola zajęte przez danego gracza.

Napisz procedurę `hex : int → (int * int) list → (bool * int)`, która na podstawie wielkości planszy n , oraz listy współrzędnych (rząd, pozycja w rzędzie) kolejno zajmowanych przez graczy pól określi, czy wygrał gracz biały (`true`), czy czarny (`false`) i w którym ruchu.

12. Dana jest (w postaci tablicy) permutacja zbioru $\{0, 1, \dots, n - 1\}$. W tej tablicy wolno nam zamieniać miejscami sąsiadujące elementy. Napisz procedurę `permutacja : int array → int`, która obliczy minimalną liczbę takich zamian potrzebnych do przekształcenia danej permutacji w jeden cykl.

Na przykład, dla permutacji `[1; 2; 0; 9; 3; 6; 5; 4; 7; 8]` wynikiem jest 2. Wystarczy zamienić np. elementy na pozycjach 2 i 3, oraz 6 i 7, a otrzymamy `[1; 2; 9; 0; 3; 6; 4; 5; 7; 8]` Podaj złożoność czasową i pamięciową swojego rozwiązania.

13. [XIV OI, Biura] W firmie pracuje n pracowników ponumerowanych od 1 do n . Każdy pracownik otrzymał służbowy telefon, w którym ma zapisane numery telefonów niektórych swoich współpracowników (a wszyscy ci współpracownicy mają w swoich telefonach zapisany jego numer). W związku z dynamicznym rozwojem firmy zarząd postanowił przenieść siedzibę firmy do nowych biurowców. Postanowiono, że każda para pracowników, którzy będą pracować w różnych budynkach, powinna znać (nawzajem) swoje numery telefonów, tzn. powinni oni mieć już zapisane nawzajem swoje numery w służ-

bowych telefonach komórkowych. Równocześnie, zarząd postanowił zająć jak największą liczbę biurów.

Napisz procedurę `biura : int -> int * int list -> int`, która na podstawie liczby pracowników n oraz listy par pracowników posiadających nawzajem swoje numery telefonów $l = [(a_1, b_1); \dots; (a_m, b_m)]$, wywołana jako `biura n l`, obliczy liczbę biurów.

Rozwiązując to zadanie przyjmij, że $m = O(n)$.

14. [ONTAK 2007] Wyobraźmy sobie ciąg zer i jedynek (x_1, x_2, \dots, x_n) . Mamy dany ciąg trójek postaci (i, j, s) , gdzie i i j są indeksami w ciągu (x_i) , $1 \leq i \leq j \leq n$, natomiast $s = (\sum_{k=i}^j x_k) \bmod 2$. Niestety dany ciąg trójek jest sfałszowany — nie istnieje ciąg (x_i) , który by go spełniał. Twoim zadaniem jest znalezienie takiego k , że dla pierwszych k danych trójek istnieje jeszcze spełniający je ciąg, a dla $k+1$ już nie. Napisz procedurę `przedzialy : int -> (int * int * int) list -> int`, która dla danego n oraz ciągu trójek oblicza takie k .

Wytyczne dla prowadzących ćwiczenia

Ad. 5 To zadanie można rozwiązywać zarówno stosując przeszukiwanie grafów, jak i drzewa find-union. (Wynik jest równy liczbie spójnych składowych.) Ponieważ o przeszukiwaniu grafów nic jeszcze nie było, należy zastosować drzewa find-union.

Ad. 6 Wynik jest równy liczbie spójnych składowych w dopełnieniu danego grafu. Nie należy jednak konstruować takiego dopełnienia, gdyż byłby to koszt czasowy rzędu $O(n^2)$. Należy znaleźć wierzchołek o minimalnym stopniu (jest on ograniczony przez stałą) i skleić z nim wszystkie wierzchołki, które nie są z nim incydentne. W ten sposób uzyskujemy graf stałej wielkości, który dalej przetwarzamy jakkolwiek, choćby w czasie $O(n^2)$.

Ad. 14 Oznaczmy przez p_i sumę (modulo 2) prefiksu ciągu (x_i) złożonego z pierwszych i elementów, $p_i = (\sum_{k=1}^i x_k) \bmod 2$. W szczególności $p_0 = 0$. Trójka (i, j, s) oznacza, że $p_j = (p_{i-1} + s) \bmod 2$.

Dla każdego $i = 0, 1, \dots, n$ mamy dwa wierzchołki, jeden reprezentujący fakt, że $p_i = 0$ i jeden reprezentujący fakt, że $p_i = 1$. Każda trójka to przyczynek do sklejania ze sobą dwóch par wierzchołków, które muszą być sobie równoważne. Jeżeli po którymś z kolei sklejeniu wierzchołki $p_j = 0$ i $p_j = 1$ będą w tej samej klasie abstrakcji, to nie istnieje ciąg (x_i) spełniający przetworzone trójki.

Wykład 15. Logika Hoare'a — dowodzenie poprawności programów imperatywnych

W tym wykładzie zajmiemy się specyfikacjami i dowodzeniem poprawności programów imperatywnych. Zaczniemy od kilku podstawowych pojęć.

15.1 Kilka podstawowych pojęć

Większość z opisanych poniżej pojęć poznaliśmy już w odniesieniu do programów funkcyjnych. Określmy teraz co oznaczają one w odniesieniu do programów imperatywnych. (Jednocześnie powinno to wyjaśnić ich nazwy.)

- W przypadku programów imperatywnych istotne jest w jaki sposób zmieniają one **stan** systemu, czyli wartości zmiennych.
- **Asercja** to warunek odnoszący się do stanu w danym momencie obliczeń / w określonym miejscu w programie, który powinien być spełniony.
- **Warunek końcowy** to asercja odnosząca się do stanu po wykonaniu danego programu. Warunek końcowy określa stany systemu, do jakich powinien prowadzić program.
- **Warunek początkowy** to asercja odnosząca się do stanu przed wykonaniem danego programu. Warunek początkowy wyraża założenia, jakie powinien spełniać stan, w którym uruchamiamy program.
- Na potrzeby niniejszego wykładu przyjmujemy, że **specyfikacja** danego programu składa się z warunku początkowego i końcowego.
- Program jest **częściowo poprawny** (względem danej specyfikacji), jeżeli dla każdego stanu spełniającego warunek początkowy, program uruchomiony w tym stanie, o ile kończy działanie (bez błędu), to kończy je w stanie spełniającym warunek końcowy.
- Program ma **własność stopu**, jeżeli dla każdego stanu spełniającego warunek początkowy, program uruchomiony w tym stanie kończy działanie (bez błędu).
- Program jest **całkowicie poprawny** (względem danej specyfikacji), jeżeli dla każdego stanu spełniającego warunek początkowy, program uruchomiony w tym stanie kończy działanie i to w stanie spełniającym warunek końcowy. Inaczej mówiąc:

$$\text{Całkowita poprawność} = \text{Częściowa poprawność} + \text{własność stopu}$$

15.2 Formalizmy do specyfikacji i weryfikacji programów imperatywnych

Powstało wiele formalizmów służących do dowodzenia poprawności programów imperatywnych, m.in.:

- logika Hoare'a [Hoa69, Apt81, BA05, Dah92] — którą zajmiemy się w tym wykładzie,
- transformatory predykatów Dijkstry [Dij85, BA05] — służą one do wyznaczania najsłabszego warunku wstępnego, gwarantującego zakończenie obliczeń i spełnienie warunku końcowego,

- logika algorytmiczna [MS92, MS87] — w dużym skrócie, logika algorytmiczna stanowi rozszerzenie języka predykatów pierwszego rzędu o formuły postaci: program, warunek końcowy; taka formuła reprezentuje najslabszy warunek początkowy gwarantujący zakończenie programu i spełnienie warunku końcowego; co więcej, takie formuły mogą być fragmentami większych formuł.

15.3 While-programy

W punkcie tym przedstawimy prosty fragment imperatywnego Ocamlu, który będziemy nazywać *while-programami*. Dalsze rozważania ograniczymy do while-programów.

Dla uproszczenia zakładamy, że wszystkie „zmienne” są referencjami do liczb całkowitych, oraz że każdy identyfikator reprezentuje inną referencję. Rozszerzenie do referencji do innych typów prostych nie jest trudne, ale komplikuje technikalnia. Natomiast wprowadzenie aliasingu, lub imperatywnych typów złożonych, takich jak tablice czy struktury wskaźnikowe, istotnie komplikuje problem specyfikacji i weryfikacji programów [Apt81, CO81, BS95, Kub00, Kub03].

```

⟨while-program⟩ ::= ⟨instrukcja⟩ | ⟨instrukcja⟩ ; ⟨while-program⟩
⟨instrukcja⟩    ::= begin ⟨while-program⟩ end |
                  if ⟨warunek⟩ then ⟨instrukcja⟩ else ⟨instrukcja⟩ |
                  while ⟨warunek⟩ do ⟨instrukcja⟩ |
                  ⟨referencja⟩ := ⟨wyrażenie⟩ |
                  ()
⟨wyrażenie⟩    ::= ...
⟨warunek⟩     ::= ...
⟨identyfikator⟩ ::= ...

```

[[Dodefiniować składnię wyrażeń, warunków i referencji.]]

15.4 Semantyka while-programów

Skoro wszystkie zmienne są typu `int`, to stan systemu możemy reprezentować jako wartościowanie zmiennych, funkcja przypisująca nazwom zmiennych ich wartości:

$$Stan = [Id \rightarrow \mathbb{Z}]$$

Semantykę while-programu możemy reprezentować jako funkcję częściową ze stanów w stany. Częściowość wynika stąd, że obliczenia mogą się zapętlać lub nie udawać z innych przyczyn.

$$\llbracket \text{while-program} \rrbracket : Stan \mapsto Stan$$

Semantyka wyrażenia, to funkcja ze stanu w liczby całkowite:

$$\llbracket \text{wyrażenie} \rrbracket : Stan \mapsto \text{int}$$

Semantyka warunku, to funkcja ze stanu w wartości logiczne:

$$\llbracket \text{warunek} \rrbracket : Stan \mapsto \text{bool}$$

$$\begin{aligned}
\llbracket () \rrbracket(S) &= S \\
\llbracket r := w \rrbracket(S) &= \text{fun } i \rightarrow \text{if } i = r \text{ then } \llbracket w \rrbracket(S) \text{ else } S(i) \\
\llbracket P_1; P_2 \rrbracket(S) &= \llbracket P_2 \rrbracket(\llbracket P_1 \rrbracket(S)) \\
\llbracket \text{if } w \text{ then } P_1 \text{ else } P_2 \rrbracket(S) &= \\
&= \text{if } \llbracket w \rrbracket(S) \text{ then } \llbracket P_1 \rrbracket(S) \text{ else } \llbracket P_2 \rrbracket(S) \\
\llbracket \text{if } w \text{ then } P \rrbracket &= \llbracket \text{if } w \text{ then } P \text{ else } () \rrbracket \\
\llbracket \text{while } w \text{ do } P \rrbracket(S_0) &= S_n, \text{ gdzie: } \text{not } \llbracket w \rrbracket(S_n) \\
&\llbracket w \rrbracket(S_i), S_{i+1} = \llbracket P \rrbracket(S_i) \text{ dla } 0 \leq i < n \\
&\text{w p.p. nieokreślone}
\end{aligned}$$

[[Rozwinąć równania dla wyrażeń i warunków logicznych.]]

15.5 Formuły logiki Hoare'a

Formuły Hoare'owskie to trójki postaci:

$$\{\varphi\}P\{\psi\}$$

gdzie:

- φ to warunek początkowy,
- P to while-program,
- ψ to warunek końcowy.

Formuła taka wyraża częściową poprawność programu P względem φ i ψ .

Modelami dla formuł Hoare'a są dziedziny algorytmiczne. Dla konkretnej dziedziny algorytmicznej i stanu początkowego wykonanie programu jest już jednoznacznie określone. O prawdziwości formuły Hoare'a możemy mówić w kontekście konkretnej dziedziny algorytmicznej. W dalszych rozważaniach zakładamy, że dziedzina algorytmiczna jest taka jak w Ocamlu.

Jeśli A jest zbiorem formuł (pierwszego rzędu) dotyczących dziedziny algorytmicznej, to $A \models \{\varphi\}P\{\psi\}$ oznacza, że $\{\varphi\}P\{\psi\}$ jest prawdziwe dla wszystkich dziedzin algorytmicznych, w których spełnione są formuły z A .

15.6 System dowodzenia

- Aksjomaty:

– instrukcji pustej:

$$A \vdash \{\varphi\} () \{\varphi\}$$

– instrukcji przypisania

$$A \vdash \{\varphi[x/w]\} x := w \{\varphi\}$$

Reguły:

- osłabiania.

$$\frac{A \models \varphi \implies \varphi', \quad A \vdash \{\varphi'\} P \{\psi'\}, \quad A \models \psi' \implies \psi}{A \vdash \{\varphi\} P \{\psi\}}$$

– średnika:

$$\frac{A \vdash \{\varphi\} P \{\xi\}, \quad A \vdash \{\xi\} Q \{\psi\}}{A \vdash \{\varphi\} P; Q \{\psi\}}$$

– instrukcji warunkowej:

$$\frac{A \vdash \{\varphi \wedge \alpha\} P \{\psi\}, \quad A \vdash \{\varphi \wedge \neg\alpha\} Q \{\psi\}}{A \vdash \{\varphi\} \text{ if } \alpha \text{ then } P \text{ else } Q \{\psi\}}$$

– pętli while:

$$\frac{A \vdash \{\varphi \wedge \alpha\} P \{\varphi\}}{A \vdash \{\varphi\} \text{ while } \alpha \text{ do } P \text{ done } \{\varphi \wedge \neg\alpha\}}$$

- Istnieje też wzmocniona wersja logiki Hoare'a pozwalająca dowodzić pełną poprawność. Fragmenty przedstawionego systemu dowodzenia należy zastąpić następującymi regułami:

– instrukcji przypisania

$$\frac{A \models \varphi[!x/w] \implies w \text{ jest określone}}{A \vdash \{\varphi[!x/w]\} x := w \{\varphi\}}$$

– instrukcji warunkowej:

$$\frac{A \models \varphi \implies \alpha \text{ jest określone}, \quad A \vdash \{\varphi \wedge \alpha\} P \{\psi\}, \quad A \vdash \{\varphi \wedge \neg\alpha\} Q \{\psi\}}{A \vdash \{\varphi\} \text{ if } \alpha \text{ then } P \text{ else } Q \{\psi\}}$$

– pętli while:

$$\frac{A \models \varphi \implies \alpha \text{ jest określone}, \quad A \models (\varphi \wedge \mu \leq 0) \implies \neg\alpha, \quad n \text{ nie występuje w } P, \quad A \vdash \{\varphi \wedge \alpha \wedge \mu = n\} P \{\varphi \wedge \mu < n\}}{A \vdash \{\varphi\} \text{ while } \alpha \text{ do } P \text{ done } \{\varphi \wedge \neg\alpha\}}$$

Przykład: Oto program obliczający minimum z dwóch liczb:

```

{!x =!x0 ∧ !y =!y0}
if !x <!y then z :=!x else z :=!y
{!z = min(!x0, !y0)}

```

Korzystając z reguły dla instrukcji warunkowej otrzymujemy dwie formuły do udowodnienia:

1.

$$\{!x =!x_0 \wedge !y =!y_0 \wedge !x <!y\} z :=!x \{!z = \min(!x_0, !y_0)\}$$

Korzystając z reguły osłabiania oraz z tego, że:

$$(!x =!x_0 \wedge !y =!y_0 \wedge !x <!y) \implies !x = \min(!x_0, !y_0)$$

otrzymujemy:

$$\{!x = \min(!x_0, !y_0)\} z :=!x \{!z = \min(!x_0, !y_0)\}$$

co jest aksjomatem przypisania.

2.

$$\{!x = !x_0 \wedge !y = !y_0 \wedge \neg(!x < !y)\} \mathbf{z} := !y \{!z = \min(!x_0, !y_0)\}$$

Korzystając z reguły osłabiania oraz z tego, że:

$$(!x = !x_0 \wedge !y = !y_0 \wedge \neg(!x < !y)) \implies !y = \min(!x_0, !y_0)$$

otrzymujemy:

$$\{!y = \min(!x_0, !y_0)\} \mathbf{z} := !y \{!z = \min(!x_0, !y_0)\}$$

co jest aksjomatem przypisania.

Przykład: Algorytm Euklidesa możemy wyspecyfikować tak:

$$\{!x > 0 \wedge !x = !x_0 \wedge !y > 0 \wedge !y = !y_0\} \mathbf{E} \{!x = NWD(!x_0, !y_0)\}$$

a jego implementacja imperatywna wygląda tak:

$$E \left\{ \begin{array}{l} \text{while } !x \neq !y \text{ do} \\ I \left\{ \begin{array}{l} \text{if } !x > !y \text{ then} \\ \quad x := !x - !y \\ \text{else} \\ \quad y := !y - !x \end{array} \right. \end{array} \right.$$

Pokażemy pełną poprawność tego algorytmu. Dowód powinien się rozpocząć od zastosowania reguły dla pętli **while**. Zauważmy, że wszystkie wyrażenia i warunki są dobrze określone. Jako funkcję miary możemy przyjąć $!x + !y$, a jako niezmiennik:

$$\varphi \equiv !x > 0 \wedge !y > 0 \wedge NWD(!x, !y) = NWD(!x_0, !y_0)$$

Musimy więc pokazać, że $(\varphi \wedge !x + !y \leq 0) \implies !x = !y$, oraz:

$$\{\varphi \wedge !x \neq !y \wedge !x + !y = n\} \mathbf{I} \{\varphi \wedge !x + !y < n\}$$

Implikacja jest trywialnie spełniona, gdyż ma wewnątrz sprzeczny poprzednik. Drugą formułę dowodzimy stosując regułę dla instrukcji **if-then-else**, co daje dwa przypadki:

$$1. \{\varphi \wedge !x \neq !y \wedge !x + !y = n \wedge !x > !y\} \mathbf{x} := !x - !y \{\varphi \wedge !x + !y < n\}$$

Oslabiamy tę formułę do aksjomatu przypisania:

$$\begin{array}{l} \{!x - !y > 0 \wedge !y > 0 \wedge NWD(!x - !y, !y) = NWD(!x_0, !y_0) \wedge !x - !y + !y < n\} \\ x := !x - !y \\ \{\varphi \wedge !x + !y < n\} \end{array}$$

korzystając z implikacji:

$$\left(\begin{array}{l} \varphi \wedge !x \neq !y \wedge \\ !x + !y = n \wedge !x > !y \end{array} \right) \implies \left(\begin{array}{l} !x - !y > 0 \wedge !y > 0 \wedge \\ NWD(!x - !y, !y) = NWD(!x_0, !y_0) \wedge \\ !x - !y + !y < n \end{array} \right)$$

2. $\{\varphi \wedge !x \neq !y \wedge !x + !y = n \wedge \neg !x > !y\} y := !y - !x \{\varphi \wedge !x + !y < n\}$

Oslabiamy tę formułę do aksjomatu przypisania:

$$\begin{array}{l} \{!x > 0 \wedge !y - !x > 0 \wedge NWD(!x, !y - !x) = NWD(!x_0, !y_0) \wedge !x + !y - !x < n\} \\ y := !y - !x \\ \{\varphi \wedge !x + !y < n\} \end{array}$$

korzystając z implikacji:

$$\left(\begin{array}{l} \varphi \wedge !x \neq !y \wedge \\ !x + !y = n \wedge \neg !x > !y \end{array} \right) \implies \left(\begin{array}{l} !x > 0 \wedge !y - !x > 0 \wedge \\ NWD(!x, !y - !x) = NWD(!x_0, !y_0) \wedge \\ !x + !y - !x < n \end{array} \right)$$

□

Przykład: Algorytm mnożenia liczb naturalnych możemy wyspecyfikować jako:

$$\{!x = !x_0 \wedge !x \geq 0 \wedge !y = !y_0 \wedge !y \geq 0\} P \{!z = !x_0 \cdot !y_0\}$$

Jego implementacja zaś wygląda następująco:

$$P \left\{ \begin{array}{l} z := 0; \\ \text{while } !x > 0 \text{ do} \\ \quad \left\{ \begin{array}{l} \text{if } !x \bmod 2 = 0 \text{ then begin} \\ \quad x := !x / 2; \\ \quad y := !y \cdot 2 \\ \quad \text{end else begin} \\ \quad x := !x - 1; \\ \quad z := !z + !y \\ \quad \text{end} \end{array} \right. \\ \text{done} \end{array} \right.$$

Pokażemy jego pełną poprawność. Na początek zauważmy, że wszystkie wyrażenia oraz warunki w instrukcji `if-then-else` i pętli `while` są zawsze dobrze określone. Tak więc tylko w przypadku reguły dla pętli `while` będziemy mieli dodatkowe przesłanki do pokazania.

Dowód poprawności musimy zacząć od wymyślenia niezmiennika pętli `while`:

$$\varphi \equiv !x \cdot !y + !z = !x_0 \cdot !y_0 \wedge !x \geq 0 \wedge !y \geq 0$$

Z reguły dla średnika uzyskujemy dwie formuły do udowodnienia:

$$\{!x = !x_0 \wedge !x \geq 0 \wedge !y = !y_0 \wedge !y \geq 0\} z := 0 \{\varphi\}$$

$$\{\varphi\} W \{!z = !x_0 \cdot !y_0\}$$

Pierwszą z nich możemy osłabić do aksjomatu przypisania:

$$\{!x \cdot !y + 0 = !x_0 \cdot !y_0 \wedge !x \geq 0 \wedge !y \geq 0\} z := 0 \{\varphi\}$$

Zauważmy, że $(\varphi \wedge !x \leq 0) \implies (!x = 0 \wedge !z = !x_0 \cdot !y_0)$. Możemy więc osłabić drugą formułę do:

$$\{\varphi\} \text{ W } \{\varphi \wedge !x \leq 0\}$$

i zastosować regułę dla pętli **while**.

Musimy teraz wybrać funkcję miary. Dobrym kandydatem jest po prostu $!x$, gdyż w każdym kroku pętli maleje i nie przyjmuje wartości ujemnych, $(\varphi \wedge !x \leq 0) \implies \neg(!x > 0)$. Pozostało do pokazania:

$$\{\varphi \wedge !x > 0 \wedge !x = n\} \text{ I } \{\varphi \wedge !x < n\}$$

Stosując regułę instrukcji warunkowej dostajemy dwie formuły:

$$1. \{\varphi \wedge !x > 0 \wedge !x = n \wedge !x \bmod 2 = 0\} \text{ x := x/2; y := y \cdot 2 } \{\varphi \wedge !x < n\}$$

Korzystając z reguły dla średnika sprowadzamy dowód tej formuły do aksjomatu przypisania:

$$\begin{aligned} &\{!x \cdot 2 \cdot !y + !z = !x_0 \cdot !y_0 \wedge !x \geq 0 \wedge 2 \cdot !y \geq 0 \wedge !x < n\} \\ &y := y \cdot 2 \\ &\{\varphi \wedge !x < n\} \end{aligned}$$

oraz formuły:

$$\begin{aligned} &\{\varphi \wedge !x > 0 \wedge !x = n \wedge !x \bmod 2 = 0\} \\ &x := x/2 \\ &\{!x \cdot 2 \cdot !y + !z = !x_0 \cdot !y_0 \wedge !x \geq 0 \wedge 2 \cdot !y \geq 0 \wedge !x < n\} \end{aligned}$$

Ją zaś możemy osłabić korzystając z implikacji:

$$\begin{aligned} &(\varphi \wedge !x > 0 \wedge !x = n \wedge !x \bmod 2 = 0) \implies \\ &\implies ((!x/2) \cdot 2 \cdot !y + !z = !x_0 \cdot !y_0 \wedge !x/2 \geq 0 \wedge 2 \cdot !y \geq 0 \wedge !x/2 < n) \end{aligned}$$

do aksjomatu przypisania:

$$\begin{aligned} &\{(!x/2) \cdot 2 \cdot !y + !z = !x_0 \cdot !y_0 \wedge !x/2 \geq 0 \wedge 2 \cdot !y \geq 0 \wedge !x/2 < n\} \\ &x := x/2 \\ &\{!x \cdot 2 \cdot !y + !z = !x_0 \cdot !y_0 \wedge !x \geq 0 \wedge 2 \cdot !y \geq 0 \wedge !x < n\} \end{aligned}$$

$$2. \{\varphi \wedge !x > 0 \wedge !x = n \wedge !x \bmod 2 \neq 0\} \text{ x :=!x - 1; z :=!z + y } \{\varphi \wedge !x < n\}$$

Dowód tej formuły przebiega podobnie do poprzedniej. Najpierw musimy skorzystać z reguły dla średnika, sprowadzając ją do aksjomatu przypisania:

$$\{!x \cdot !y + !z + !y = !x_0 \cdot !y_0 \wedge !x \geq 0 \wedge !y \geq 0 \wedge !x < n\} \text{ z :=!z + y } \{\varphi \wedge !x < n\}$$

oraz formuły:

$$\begin{aligned} &\{\varphi \wedge !x > 0 \wedge !x = n \wedge !x \bmod 2 \neq 0\} \\ &x :=!x - 1 \\ &\{!x \cdot !y + !z + !y = !x_0 \cdot !y_0 \wedge !x \geq 0 \wedge !y \geq 0 \wedge !x < n\} \end{aligned}$$

Tę zaś formułę możemy osłabić korzystając z implikacji:

$$\begin{aligned} &(\varphi \wedge !x > 0 \wedge !x = n \wedge !x \bmod 2 \neq 0) \implies \\ &\implies ((!x - 1) \cdot !y + !z + !y = !x_0 \cdot !y_0 \wedge !x - 1 \geq 0 \wedge !y \geq 0 \wedge !x - 1 < n) \end{aligned}$$

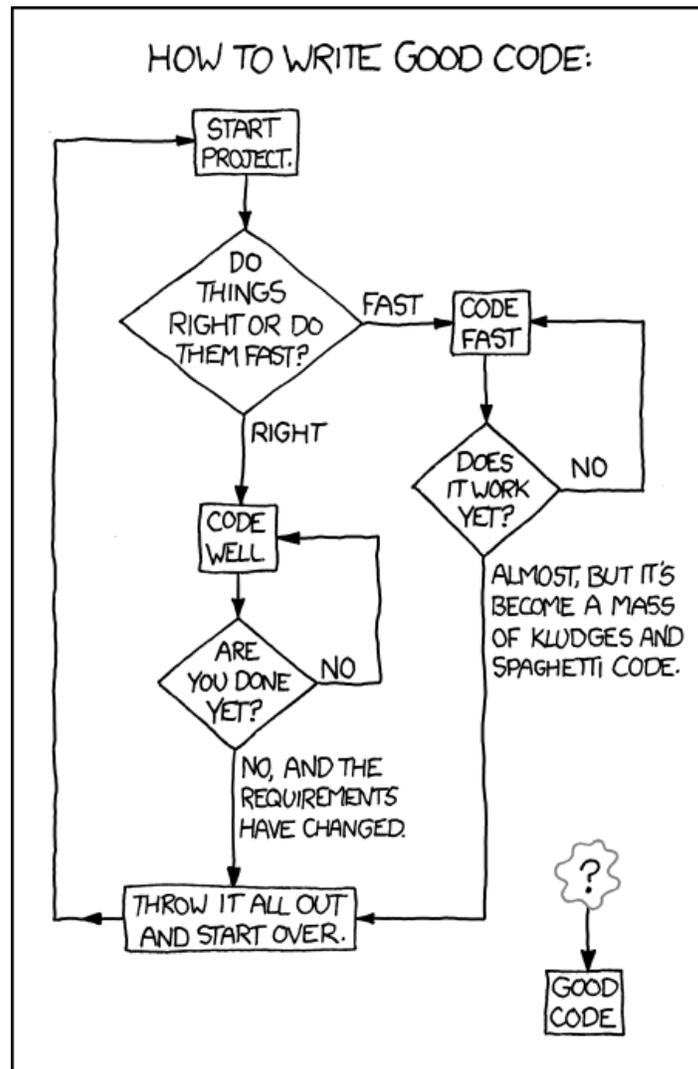
do aksjomatu przypisania:

$$\begin{aligned} & \{(!x - 1) \cdot !y + !z + !y = !x_0 \cdot !y_0 \wedge !x - 1 \geq 0 \wedge !y \geq 0 \wedge !x - 1 < n\} \\ & x := !x - 1 \\ & \{!x \cdot !y + !z + !y = !x_0 \cdot !y_0 \wedge !x \geq 0 \wedge !y \geq 0 \wedge !x < n\} \end{aligned}$$

□

Przykład: [[Znalezienie zera funkcji całkowitoliczbowej metodą bisekcji. Uwaga: zero nie musi istnieć.]]

15.7 Deser



<http://xkcd.com/303/>

Ćwiczenia

- Udowodnij następującą trójkę:

$$\{x = x_0, y = y_0\} y := !x - !y; x := !x - !y; y := !x + !y \{x = x_0, y = x_0\}$$

- Udowodnij poprawność programu iteracyjnego obliczającego silnię.
- Udowodnij poprawność programu iteracyjnego obliczającego liczby Fibonacciego.
- Udowodnij poprawność programu iteracyjnego obliczającego $\binom{n}{k}$
- Wyszukanie minimum w ciągu $(f(0), \dots, f(n))$.
- Określ dla każdego n , co jest wynikiem funkcji `coto` i udowodnij to.

```
let coto n =
  let x = ref 0
  and y = ref 1
  in begin
    while !y <= n do
      x := !x - !y;
      y := !y - !x
    done;
    if n mod 2 = 1 then !x else (!x)
  end;;
```

- Dana jest n -elementowa tablica a , w której znajdują się liczby ze zbioru $\{0, 1, \dots, n-1\}$. Na tablicy wolno wykonywać tylko następujące operacje:
 - `length a` — badać wielkość tablicy n ,
 - `a.(i)` — odczytywać elementy tablicy, oraz
 - `zamien(i, j)` — zamieniać elementy wartościami.

Napisz procedurę `zamiany : int array → unit`, która posortuje daną tablicę (tj. przekształci ją w permutację identycznościową) wykonując minimalną możliwą liczbę zamian.

Podaj niezmienniki pętli oraz odpowiednie funkcje miary. Nie muszą one mieć postaci formuł, ale muszą być ściśle określone. Udowodnij pełną poprawność programu.

- Podaj niezmiennik pętli i wykaż poprawność następującego programu:

```
{x = x_0 ≥ 0}
i := 0;
d := 1;
s := 1;
```

```
while !s <= !x do
  d := !d + 2;
  s := !s + !d;
  i := !i + 1
done;
{i = ⌊√x₀⌋}
```

Wykład 16. Przeszukiwanie grafów

16.1 Grafy

Grafy składają się z wierzchołków i łączących je krawędzi. Standardowo przyjmuje się, że krawędzie muszą mieć różne końce i między dwoma wierzchołkami może być co najwyżej jedna krawędź. Odrzucając te warunki możemy rozważać, odpowiednio, grafy z pętelkami lub multi-grafy. Mówiąc „grafy” mamy na myśli grafy nieskierowane, czyli takie, w których krawędzie są symetryczne. Można też rozważać grafy skierowane, w których krawędzie mają określony kierunek. Czasami krawędź nieskierowaną utożsamia się z parą krawędzi skierowanych prowadzących w obu kierunkach.

Zaimplementujmy grafy nieskierowane. Możemy to zrobić w postaci modułu o sygnaturze:

```
module type GRAPH_LABELS =
  sig
    type label
  end;;

module type GRAPH =
  sig
    (* Etykiety krawędzi. *)
    type label

    (* Typ grafów o wierzchołkach typu 'a. *)
    type graph

    (* Pusty graf. *)
    val init : int ->graph

    (* Rozmiar grafu *)
    val size : graph ->int

    (* Dodanie krawędzi skierowanej łączącej dwa wierzchołki. *)
    val insert_directed_edge : graph ->int ->int ->label ->unit

    (* Dodanie krawędzi nieskierowanej (dwukierunkowej) łączącej dwa wierzchołki. *)
    val insert_edge : graph ->int ->int ->label ->unit

    (* Lista incydencji danego wierzchołka. *)
    val neighbours : graph ->int ->(int*label) list
  end;;
```

Stosowane są dwa rodzaje struktur danych do reprezentowania grafów: macierze sąsiedztwa i listy incydencji (sąsiedztwa). Macierze sąsiedztwa możemy stosować tam, gdzie wierzchołki są ponumerowane kolejnymi liczbami całkowitymi. Jeśli w grafie mamy n wierzchołków, to macierz sąsiedztwa ma postać macierzy $n \times n$ wartości logicznych. To, czy istnieje krawędź łącząca wierzchołki v i w jest określone w wierszu v i kolumnie w macierzy. W przypadku grafów nieskierowanych macierz sąsiedztwa jest symetryczna. Zaletą macierzy sąsiedztwa jest to, że możemy w czasie stałym sprawdzać, czy dwa wierzchołki są połączone i możliwość zmiany tego. Do wad należy zaliczyć wielkość macierzy oraz to, że wyliczenie wszystkich sąsiadów danego wierzchołka zajmuje zawsze czas $\Theta(n)$.

Lista incydencji to lista wszystkich sąsiadów danego wierzchołka. Jeżeli wierzchołki są ponumerowane liczbami całkowitymi, to możemy reprezentować cały graf jako tablicę takich list. Zaletą takiej struktury danych jest to, że zajmuje ona pamięć rzędu $\Theta(n + m)$, gdzie m jest liczbą krawędzi. Również wyliczenie sąsiadów danego wierzchołka staje się trywialne. Właśnie taką reprezentację zastosujemy tutaj.

```

module Graph = functor (L : GRAPH_LABELS) ->
  (struct
    (* Etykiety krawędzi. *)
    type label = L.label

    (* Typ grafów - tablica list sąsiedztwa. *)
    type graph = {n : int; e : (int*label) list array}

    (* Pusty graf. *)
    let init s = {n = s; e = Array.make s []}

    (* Rozmiar grafu. *)
    let size g = g.n

    (* Dodanie krawędzi skierowanej łączącej dwa wierzchołki. *)
    let insert_directed_edge g x y l =
      assert ((x<>y) && (x >= 0) && (x <size g) && (y >= 0) && (y <size g) &&
(List.filter (fun (v,_) ->v=y) g.e.(x) = []));
      g.e.(x) <- (y,l)::g.e.(x)

    (* Dodanie krawędzi łączącej dwa (istniejące) wierzchołki. *)
    let insert_edge g x y l =
      insert_directed_edge g x y l;
      insert_directed_edge g y x l

    (* Lista incydencji danego wierzchołka. *)
    let neighbours g x =
      g.e.(x)
  end : GRAPH with type label = L.label);;

```

16.2 Przeszukiwanie grafów

Problem przeszukiwania grafu polega na tym, że chcemy obejść wierzchołki grafu w określonej kolejności. Przede wszystkim, chcemy obejść po kolei spójne składowe grafu. Co do kolejności obejścia wierzchołków w obrębie jednej składowej, to przyjrzymy się temu dalej. Parametrem procedury przeszukującej graf jest procedura (oznaczona poniżej jako `visit`), którą należy wywołać dla każdego odwiedzanego wierzchołka. Każdą składową chcemy pokolorować innym „kolorem”. Ów kolor to liczba całkowita, od 0 wzwyż — jest on parametrem procedury `visit`. Wynikiem procedury przeszukującej graf jest liczba znalezionych spójnych składowych. Przeszukiwanie grafu możemy zrealizować jako następujący funktor:

```

module GraphSearch (Q : QUEUE) (G : GRAPH) =
  struct

```

```

open G
let search visit g =
  let visited = Array.make (size g) false
  in
    let walk x color =
      let q = ref Q.empty
      in
        begin
          q := Q.insert !q x;
          visited.(x) <- true;
          while not (Q.is_empty !q) do
            let v = Q.front !q
            in begin
              visit v color;
              q := Q.remove !q;
              List.iter
                (fun (y,_) ->
                  if not (visited.(y)) then begin
                    q := Q.insert !q y;
                    visited.(y) <- true
                  end)
                (neighbours g v);
            end
          done
        end
    in
      (* Kolor *)
      let k = ref 0
      in
        in
          for i = 0 to size g do
            if not visited.(i) then begin
              (* Kolejna składowa. *)
              walk i (!k);
              k := !k+1
            end;
          done;
          !k
        end;
      end;
end;;

```

Oznaczmy przez n liczbę wierzchołków, a przez m liczbę krawędzi w grafie. Zauważmy, że każdy wierzchołek, poza wierzchołkami dla których wywoływana jest procedura `search`, jest rozpatrywany tyle razy, ile wychodzi z niego krawędzi, ale tylko raz zostanie wstawiony do kolejki `q`. Tak więc rozmiar kolejki nie przekroczy liczby wierzchołków. Stąd złożoność pamięciowa (nie licząc samego grafu) jest rzędu $\Theta(n)$. Złożoność czasowa zależy oczywiście od tego, w jakim czasie działają operacje na kolejce. Zakładając, że działają w czasie stałym (zamortyzowanym) uzyskujemy złożoność czasową $\Theta(n + m)$.

16.3 Algorytm przeszukiwania wszerz BFS

Idea algorytmu przeszukiwania grafów wszerz przypomina rozprzestrzenianie się ognia w trakcie wypalania traw na łąkach, czy pożar prerii. Jeśli ogień dojdzie do jakiegoś miejsca, to

roprzestrzenia się na sąsiednie miejsca, na których jest jeszcze niewypalona trawa. W ten sposób przyłożenie ognia w jednym punkcie, spala cały „spójny obszar” suchej trawy.

Źródłem nazwiemy każdy taki wierzchołek, który odwiedzamy jako pierwszy w danej spójnej składowej. Odwiedzenie jednego wierzchołka pociąga za sobą odwiedzenie jako kolejnych, jego (nieodwiedzonych jeszcze) sąsiadów. Istotna jest przy tym kolejność odwiedzania: najpierw odwiedzane jest źródło potem jego sąsiedzi, potem ich sąsiedzi itd. Tak więc wierzchołki są odwiedzane w kolejności rosnącej odległości od źródła.

Do pamiętania wierzchołków, które oczekują na odwiedzenie, ale nie zostały jeszcze odwiedzone, wykorzystujemy kolejkę FIFO.

```
module BFS = GraphSearch (Fifo);;
```

Operacje na kolejce FIFO są wykonywane w stałym czasie zamortyzowanym. Tak więc złożoność czasowa przeszukiwania wszcz to $\Theta(n + m)$.

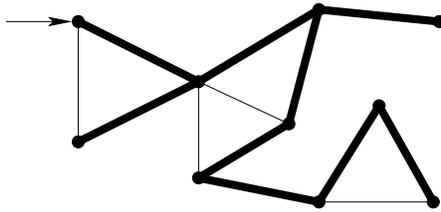
Algorytm przeszukiwania wszcz ma jeszcze jedną ciekawą właściwość. Dla każdego wywołania `search x`, wierzchołki są odwiedzane zgodnie z rosnącymi odległościami od wierzchołka `x`. Można pokazać, że w każdej chwili, w kolejce `q` znajdują się albo wierzchołki położone w tej samej odległości `l` od wierzchołka `x`, albo najpierw wierzchołki położone w odległości `l` a dalej w odległości `l + 1`. Niewątpliwie jest tak na samym początku, gdy w kolejce jest tylko wierzchołek `x`. W każdym kroku pętli `while` wyjmowany jest z początku kolejki wierzchołek położony w odległości `l` od wierzchołka `x`, a na koniec kolejki wstawiani są ci jego sąsiedzi, którzy nie zostali jeszcze rozpatrzeni, a więc znajdują się w odległości `l + 1` od wierzchołka `x`.

16.4 Algorytm przeszukiwania w głąb DFS

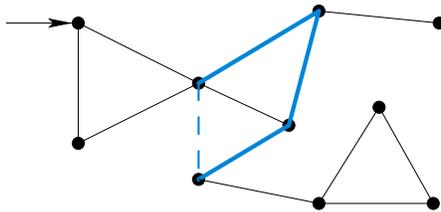
Algorytm przeszukiwania w głąb jest algorytmem przeszukiwania z nawrotami. Procedurę przeszukującą wywołujemy dla źródła. Następnie jest ona rekurencyjnie wywoływana dla jego sąsiadów, ich sąsiadów itd. Przy tym, jeżeli jest ona wywoływana dla danego wierzchołka po raz kolejny, to nic się nie dzieje. Implementacja tej procedury nie musi być rekurencyjna. Może być iteracyjna, jeżeli zastosujemy stos, czyli kolejkę LIFO (ang. *last in, first out*).

```
module Lifo : QUEUE =
  struct
    exception EmptyQueue
    type 'a queue = 'a list
    let empty = []
    let is_empty q = q = []
    let insert q x = x::q
    let front q =
      if q = [] then raise EmptyQueue
      else hd q
    let remove q =
      if q = [] then raise EmptyQueue
      else tl q
  end;;
```

```
module DFS = GraphSearch (Lifo);;
```



Rysunek 3: Drzewo rozpinające wyznaczone przez algorytm DFS. Strzałką zaznaczono źródło.



Rysunek 4: Przykład krawędzi spoza drzewa rozpinającego DFS oraz cyklu przez nią rozpiętego.

Operacje na kolejce LIFO wykonywane są w stałym czasie, tak więc złożoność czasowa tego algorytmu jest rzędu $\Theta(n + m)$.

Algorytm ten ma wiele ciekawych własności. Pozwala on na znalezienie w grafie cykli prostych, a także na podział grafu na dwuspójne składowe. Wyobraźmy sobie drzewa zbudowane z tych krawędzi, które łączą odwiedzone wierzchołki i wierzchołki, które przy tej okazji są wstawiane do kolejki. Korzeniem drzewa jest źródło. Są to drzewa rozpinające spójne składowe grafu. Zastanówmy się, jak mogą być położone względem takiego drzewa pozostałe krawędzie grafu.

W ogólności, krawędź spoza drzewa rozpinającego może być położona na dwa sposoby:

- może ona łączyć dwie różne gałęzie drzewa,
- może ona prowadzić w górę/dół drzewa.

Okazuje się jednak, że w przypadku drzewa rozpinającego wyznaczonego przez algorytm DFS pierwszy przypadek nie jest możliwy. Gdyby tak było, to w trakcie obchodzenia pierwszej z tych gałęzi powinniśmy wstawić do kolejki wierzchołki leżący na drugim końcu rozpatrywanej krawędzi. Wówczas stałaby się ona częścią drzewa rozpinającego wyznaczonego przez algorytm DFS. Przykład drzewa rozpinającego DFS przedstawiono na rys. 3.

Tak więc, krawędzie spoza drzewa muszą prowadzić w górę/dół drzewa rozpinającego. Z każdą taką krawędzią związany jest cykl, złożony z niej samej oraz ścieżki w drzewie łączącej jej końce. Przykład takiej krawędzi i cyklu przedstawiono na rys. 4. Okazuje się, że dowolny cykl prosty w grafie możemy przedstawić jednoznacznie, jako zbiór krawędzi spoza drzewa rozpinającego DFS, wykracza to jednak poza ramy tego wykładu.

16.5 Sortowanie topologiczne

Rozważmy teraz następujący problem. Mamy do wykonania n różnych czynności. Niektóre z nich muszą być wykonane wcześniej, a niektóre później. Mamy danych m par zależności

postaci: czynność a musi być wykonana wcześniej niż czynność b . Należy ustalić taką kolejność wykonywania czynności, żeby wszystkie zależności były spełnione lub stwierdzić, że nie jest to możliwe.

Dane możemy przedstawić jako graf skierowany. Czynności to wierzchołki, a zależności to krawędzie. Fakt, że a należy wykonać wcześniej, a b później reprezentujemy jako krawędź $a \rightarrow b$.

Do rozwiązania tego problemu możemy użyć zmodyfikowanej wersji algorytmu DFS. Pierwsza modyfikacja polega na tym, że możemy przechodzić wzdłuż krawędzi tylko zgodnie z ich kierunkiem. Druga modyfikacja polega na tym, że każdy wierzchołek odwiedzamy dwa razy — najpierw do niego „*wchodzimy*”, następnie odwiedzamy wszystkich jego sąsiadów, a na koniec z niego „*wychodzimy*”. W ten sposób obchodzimy drzewo rozpinające DFS, przechodząc wzdłuż każdej krawędzi dwa razy. (Takie obejście drzewa nazywa się *kartezjańskim*).

Zastanówmy się, jak mogą być położone krawędzie spoza drzewa rozpinającego DFS. Krawędzie drzewa rozpinającego DFS są zorientowane od korzenia w dół. Jeśli więc spotkamy krawędź prowadzącą w górę drzewa, to mamy cykl w grafie i rozwiązanie nie istnieje. Krawędź taką poznamy po tym, że prowadzi do wierzchołka, do którego weszliśmy, ale z którego jeszcze nie wyszliśmy.

Jeśli napotkamy krawędź prowadzącą do wierzchołka, z którego już wyszliśmy, to jest to wierzchołek leżący na innej gałęzi drzewa rozpinającego. Reprezentowaną przez niego czynność należy wykonać później. Dotyczy to również wszystkich innych wierzchołków, do których można z niego dojść. Zauważmy, że one również zostały już odwiedzone i z nich wyszliśmy.

Jeżeli napotkamy krawędź prowadzącą do wierzchołka, którego nie odwiedzaliśmy jeszcze, to do niego wchodzimy i obchodzimy wierzchołki do których da się z niego dojść. Zauważmy, że wejdziemy do niego później, ale wyjdziemy z niego wcześniej, niż z wierzchołka, w którym jesteśmy.

Tak więc, z wszystkich wierzchołków reprezentujących czynności, które należy wykonać później, wyjdziemy wcześniej niż z danego wierzchołka. Wystarczy więc wykonywać czynności w odwrotnej kolejności wychodzenia z wierzchołków. Implementację takiego rozwiązania pozostawiamy czytelnikowi jako ćwiczenie (lub zadanie na laboratorium).

[[Dodać implementację, zmodyfikować odwiedzanie tak, aby alg. Dijkstry był instancją ogólnego algorytmu.]]

16.6 Algorytm Dijkstry

Algorytm Dijkstry służy do wyznaczania odległości od zadanego wierzchołka do pozostałych wierzchołków, w grafie nieskierowanym, w którym krawędzie mają długości. Zakładamy, że długości krawędzi są nieujemnymi liczbami rzeczywistymi.

Przeszukiwanie rozpoczynamy od tego wierzchołka, od którego odległości chcemy wyznaczyć — nazwijmy go *źródłem*. Odwiedzane wierzchołki trzymamy w kolejce priorytetowej, a priorytety odpowiadają odległości od źródła. Wierzchołki zaznaczamy jako odwiedzone dopiero po wyjęciu z kolejki, a wyjmując, upewniamy się, czy wcześniej już danego wierzchołka nie odwiedziliśmy. W ten sposób wierzchołki mogą być wstawiane do kolejki priorytetowej wielokrotnie, jednak nie więcej niż łącznie m razy. Może się jednak zdarzyć, że wstawiając wierzchołek do kolejki po raz kolejny, nadamy mu mniejszą wagę i zostanie on wcześniej wyjęty z kolejki. Patrząc na pierwsze wyjęcia z kolejki wierzchołków, wyjmujemy je w kolejności rosnącej odległości od źródła.

```

module FloatLabel =
  struct
    type label = float
  end;;

module FloatGraph = Graph(FloatLabel);;

module Dijkstra :
  sig
    val search : FloatGraph.graph ->int ->float array
  end = struct
    open FloatGraph

    module Order =
      struct
        type t = int * float
        let porownaj (v1,d1) (v2,d2) =
          if (d1,v1) <(d2,v2) then Wieksze
          else if (d1,v1) = (d2,v2) then Rowne
          else Mniejsze
        end
      end

    module Q = PriQueue (Order)

    let search g v =
      let visited = Array.make (size g) false
          and dist = Array.make (size g) infinity
          and q = ref Q.empty
      in begin
        q := Q.put (v,0.0) !q;
        while not (Q.is_empty !q) do
          let (v,d) = Q.getmax !q
          in
            if not visited.(v) then begin
              List.iter
                (fun (w,l) ->q := Q.put (w,d +. l) !q)
                (neighbours g v);
              dist.(v) <- d;
              visited.(v)<- true
            end;
          q := Q.removemax !q
        done;
        dist
      end
    end;;

```

Koszt czasowy operacji na kolejce priorytetowej jest rzędu $\Theta(\log n)$. Dlatego też koszt czasowy tego algorytmu jest rzędu $\Theta((n + m) \log n)$.

16.7 Algorytm Floyd-Warshalla

[[Uzupełnić]]

Ćwiczenia

1. Dany jest graf nieskierowany, którego wierzchołki są ponumerowane od 0 do $n - 1$. Napisz procedurę `path : graph → int`, która wyznacza długość (liczoną liczbą krawędzi) najdłuższej ścieżki w tym grafie, na której numery wierzchołków tworzą ciąg rosnący.
2. [II OI, zadanie *Klub Prawoskrętnych Kierowców*, PCh] Mamy dany układ ulic, leżących na prostokątnej siatce $m \times n$. Ulice są jedno- lub dwukierunkowe. Przyjmujemy, że cztery strony świata są reprezentowane za pomocą liczb całkowitych od 0 do 3:

```
let north = 0 and east = 1 and south = 2 and west = 3;;
```

Układ ulic jest dany w formie trójwymiarowej tablicy wartości logicznych `ulice.(i).(j).(k)` określa, czy z punktu o współrzędnych (i, j) można przejechać w kierunku k jedną jednostką odległości. Można założyć, że tablica ta uniemożliwia wyjechanie poza obszar $\{0, \dots, m - 1\} \times \{0, \dots, n - 1\}$.

Członkowie Klubu Prawoskrętnych Kierowców na skrzyżowaniach jeżdżą prosto lub skręcają w prawo. Sprawdź, czy z punktu $(0, 0)$ prawoskrętny kierowca może dojechać do punktu $(m - 1, n - 1)$.

3. [XIV OI, zadanie *Grzbiety i doliny*] Mamy daną mapę topograficzną terenu, w postaci prostokątnej tablicy liczb całkowitych, reprezentujących wysokości terenu. Grzbietem (doliną) nazwiemy taki obszar, który:
 - jest spójny (przyjmujemy, że kwadraty jednostkowe mapy sąsiadują ze sobą jeżeli stykają się bokami),
 - wszystkie kwadraty na tym obszarze mają tę samą wysokość,
 - jest on otoczony kwadratami o mniejszej (większej) wysokości.

Wyznacz grzbiety i doliny na mapie.

4. Dana jest prostokątna mapa wysokości górzystego terenu, w postaci prostokątnej tablicy dodatnich liczb całkowitych, o wymiarach $N \times M$. Chcemy przejść z pola o współrzędnych $(0, 0)$ na pole o współrzędnych $(N - 1, M - 1)$, ale nie chcemy wspinać się zbyt wysoko. (Możemy się przesuwac w kierunkach N, W, S, E.) Napisz procedurę `wysokosc : int array array → int`, która dla danej mapy terenu określi minimalną największą wysokość, na którą musimy wejść w trakcie podróży.
5. [XI OI, zadanie *Wyspy*, zmodyfikowane] Dana jest prostokątna mapa $n \times m$ przedstawiająca archipelag na oceanie, reprezentowana jako `bool array array`. Elementy tablicy równe `true` reprezentują ląd, a `false` wodę. Na zewnątrz mapy znajduje się ocean.

Wyspy na oceanie są wyspami rzędu 1. Na wyspach rzędu 1 mogą znajdować się jeziora rzędu 1, na nich mogą znajdować się wyspy rzędu 2 itd. Ogólnie:

- ocean ma rząd 0,
- wyspa na jeziorze (lub oceanie) rzędu k , ma rząd $k + 1$,
- jezioro na wyspie rzędu k ma rząd k .

Pola wody przylegające do siebie bokami lub rogami należą do tego samego jeziora (lub oceanu). Pola lądu przylegające do siebie bokami należą do tej samej wyspy.

Napisz procedurę `wyspa : bool array array → int`, która dla danej mapy zwróci maksymalny rząd wyspy na mapie. Jeżeli na oceanie nie ma żadnej wyspy, to procedura powinna zwrócić 0.

Przykład: 69.793 N, 108.241 W.

6. [XIV OI, zadanie *Biura*] Firma ma n pracowników. Każdy pracownik ma telefon komórkowy, a w nim telefony pewnej grupy innych pracowników. Przy tym, jeżeli pracownik A ma telefon pracownika B , to pracownik B ma numer pracownika A . Firma chce podzielić pracowników na grupy, w taki sposób żeby:

- każdych dwóch pracowników albo było w tej samej grupie, albo miało nawzajem swoje numery telefonów,
- liczba grup była maksymalna.

Wyznacz podział pracowników na grupy.

Uwaga: Oczekiwana złożoność czasowa to $O(n^2)$, ale można to zadanie rozwiązać lepiej.

7. [II OI, zadanie *Palindromy*] Podziel dane słowo (listę znaków) na minimalną liczbę palindromów parzystej długości.
8. Dany jest acykliczny graf skierowany (DAG). Jego wierzchołki są ponumerowane od 0 do n , a krawędzie są dane w postaci tablicy sąsiedztwa (kwadratowej tablicy e wartości logicznych; w grafie mamy krawędź $u \rightarrow v$ wtw., gdy $e.(u).(v)$). Powiemy, że wierzchołek u *dominuje* wierzchołek v , jeżeli dla każdego wierzchołka w , z którego można dojść do v , z u można dojść do w .

Napisz procedurę `zdominowani : bool array array -> int`, która na podstawie tablicy opisującej graf obliczy liczbę wierzchołków, dla których istnieją wierzchołki je dominujące.

9. Dany jest graf skierowany o jednym wyróżnionym wierzchołku s nazywanym *wejściem*. Powiemy, że wierzchołek u *dominuje* wierzchołek v , jeżeli każda ścieżka z s do v musi przechodzić przez u . Zauważmy, że relacja dominacji jest częściowym porządkiem.

Powiemy, że u jest *bezpośrednim dominatorem* wierzchołka v , jeżeli:

- $u \neq v$,
- u dominuje v , oraz
- każdy wierzchołek $w \neq v$, który dominuje v , dominuje też u .

Zauważmy, że relacja bezpośredniej dominacji tworzy drzewo.

Napisz procedurę, która dla danego grafu skierowanego i wierzchołka s , dla każdego wierzchołka (poza s) wyznaczy jego bezpośredniego dominatora.

10. [XIV OI, zadanie *Powódź*] Dana jest mapa topograficzna miasta położonego w kotlinie, w postaci prostokątnej tablicy typu `(int * bool) array array`. Liczby określają wysokość kwadratów jednostkowych, a wartości logiczne określają, czy dany kwadrat należy

do terenu miasta. Przyjmujemy, że teren poza mapą jest położony wyżej niż jakikolwiek kwadrat na mapie.

Miasto zostało całkowicie zalane. Żeby je osuszyć należy w kotlinie rozmieścić pewną liczbę pomp. Każda z pomp wypompowuje wodę aż do osuszenia kwadratu jednostkowego, na którym się znajduje. Osuszenie dowolnego kwadratu pociąga za sobą obniżenie poziomu wody lub osuszenie kwadratów jednostkowych, z których woda jest w stanie spłynąć do kwadratu, na którym znajduje się pompa. Woda może przepływać między kwadratami, które stykają się bokami.

Wyznacz minimalną liczbę pomp koniecznych do osuszenia miasta. Teren nie należący do miasta nie musi być osuszony. Miasto nie musi tworzyć spójnego obszaru.

11. [VIII OI, Spokojna komisja] [...]
12. [IX OI, Komiwojażer Bajtazar] [...]
13. [VIII OI, Mrówki i biedronka] [...]

Wytyczne dla prowadzących ćwiczenia

Ad. 6 Oczekujemy rozwiązania działającego w czasie $O(n^2)$, ale można to zadanie rozwiązać lepiej. Zadanie polega na wyznaczeniu spójnych składowych dopełnienia danego grafu. Zanim skonstruujemy dopełnienie danego grafu, należy go zredukować sklejając wierzchołki, które muszą być w tych samych spójnych składowych. W szczególności, jeżeli wierzchołki v i w mają stopnie mniejsze niż $\frac{n}{2}$, to są w tej samej spójnej składowej i można je skleić. W ten sposób redukujemy graf, aż liczba wierzchołków będzie równa co najwyżej $8\sqrt{m}$, gdzie m to liczba krawędzi w danym grafie. Wówczas konstrukcja dopełnienia grafu zajmuje czas rzędu $O(m)$.

Ad. 7 Spodziewamy się rozwiązania działającego w czasie $\Theta(n^2)$.

Ad. 10 Każdej pompie odpowiada przeszukiwanie wyznaczające osuszany przez nią teren. Wszystkie te przeszukiwania można połączyć w jedno, o lepszej łącznej złożoności czasowej.

16.8 Mapy

Koncepcja mapy/słownika/funkcji częściowej.

module type MAP =

sig

(* Mapa z wartości typu 'a w 'b. *)

type ('a,'b) map

(* Wyjątek podnoszony, gdy badamy wartość spoza dziedziny. *)

exception Undefined

(* Pusta mapa. *)

val empty : ('a,'b) map

(* Predykat charakterystyczny dziedziny mapy. *)

val dom : ('a,'b) map ->'a ->bool

(* Zastosowanie mapy. *)

val apply : ('a,'b) map ->'a ->'b

(* Dodanie wartości do mapy. *)

val update : ('a,'b) map ->'a ->'b ->('a,'b) map

end;;

Uwaga: Ocaml zawiera moduł Map. Implementuje on nie tylko funkcjonalność przedstawioną tutaj, ale również wiele innych procedur. Choć nazwy niektórych z nich są inne niż tutaj.

16.8.1 Prosta implementacja proceduralna

Mapa to procedura, która punktom z dziedziny przyporządkowuje wartości, a dla pozostałych punktów podnosi wyjątki.

```

module ProcMap : MAP =
  struct
    (* Mapa z wartości typu 'a w 'b. *)
    type ('a,'b) map = 'a ->'b

    (* Wyjątek podnoszony, gdy badamy wartość spoza dziedziny. *)
    exception Undefined

    (* Pusta mapa *)
    let empty = function _ ->raise Undefined
    (* Predykat charakterystyczny dziedziny mapy. *)
    let dom m x =
      try
        let _ = m x
        in true
      with Undefined ->false

    (* Zbadanie wartości w punkcie *)
    let apply m x = m x

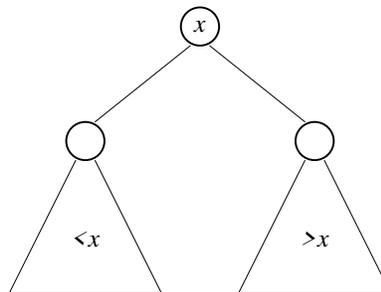
    (* Dodanie wartości do mapy. *)
    let update f a v =
      function x ->
        if a = x then v else f x
    end;;
  end

```

Jest to oczywiście implementacja nieefektywna.

16.8.2 Drzewa BST

Idea drzew BST. Do zaimplementowania drzew BST potrzebny jest porządek liniowy określony na wartościach, które mogą pojawiać się jako klucze. Ocaml na każdym typie określa porządek liniowy.



Struktura danych. Węzeł ma cztery argumenty: klucz, wartość, lewe i prawe poddrzewo.

```

module BstMap : MAP =
  struct
    (* Mapa z wartości typu 'a w 'b. *)
    type ('a,'b) map =
  end

```

```

Empty |
Node of ('a * 'b * ('a,'b) map * ('a,'b) map)

(* Wyjątek podnoszony, gdy badamy wartość spoza dziedziny. *)
exception Undefined

(* Pusta mapa to puste drzewo *)
let empty = Empty

(* Znajdź poddrzewo o zadanym kluczu *)
(* (jeśli nie ma, to wynikiem jest puste drzewo). *)
let rec find tree key =
  match tree with
  Empty -> Empty |
  Node (k,_,l,r) ->
    if key = k then tree
    else if key <k then find l key
    else find r key

(* Zastosowanie mapy. *)
let apply m k =
  match find m k with
  Empty -> raise Undefined |
  Node (_,v,_,_) -> v

(* Sprawdzenie, czy punkt należy do dziedziny *)
let dom m x =
  try
    let _ = apply m x
    in true
  with
  Undefined -> false

(* Zmiana wartości mapy w punkcie *)
let rec update tree key value =
  match tree with
  Empty -> Node (key,value,Empty,Empty) |
  Node (k,v,l,r) ->
    if key = k then
      Node (key,value,l,r)
    else if key <k then
      Node (k,v,update l key value,r)
    else
      Node (k,v,l,update r key value)
end;;

```

Pesymistyczna złożoność operacji na drzewie BST jest proporcjonalna do wysokości drzewa, czyli w najgorszym przypadku jest rzędu $O(n)$. Jeśli jednak wstawiane wartości są losowe, to średnia wysokość drzewa jest dużo mniejszego rzędu: $O(\log n)$.

Są też zrównoważone wersje drzew BST (np. AVL), które zawsze gwarantują, że wysokość drzewa jest rzędu $O(\log n)$.

Wykład 17. Back-tracking

W tym wykładzie przedstawimy technikę rozwiązywania problemów nazywaną *przeszukiwaniem z nawrotami* lub z angielska *back-tracking*. Technika ta służy do rozwiązywania problemów, do których można zastosować zasadę „dziel i rządź”, ale dany problem sprowadza się do podproblemu(-ów) na wiele sposobów. Szukając więc rozwiązania, musimy podjąć szereg decyzji jak podzielić/zmniejszyć rozpatrywany problem. Przeszukiwanie z nawrotami polega na rekurencyjnym przeszukiwaniu wszystkich możliwych ciągów takich decyzji.

Techniki tej możemy również użyć do rozwiązywania problemów optymalizacyjnych, czyli takich, gdzie nie tylko chcemy znaleźć możliwe rozwiązanie, ale rozwiązanie pod pewnym względem najlepsze. Jeżeli w trakcie przeszukiwania okazuje się, że wcześniej podjęte decyzje nie prowadzą do rozwiązania lub nie prowadzą do rozwiązania lepszego od już znanego, to z takiej gałęzi rekurencyjnych przeszukiwań można się wycofać i spróbować innej gałęzi.

17.1 Prosty generyczny back-tracking

Zdefiniujemy teraz prosty ogólny mechanizm back-tracking, dla problemów, w których dokonując wyboru, sprowadzamy problem do jednego problemu o mniejszym rozmiarze. Tak więc szukane rozwiązanie możemy traktować jak ciąg prowadzących do niego wyborów.

Zdefiniujemy funktor, który na podstawie instancji problemu znajduje jedno lub wszystkie jego rozwiązania. Instancja problemu musi definiować następujące pojęcia:

- **konfiguracja** — jest to typ, którego wartości reprezentują dokonane wybory; kolejny wybór polega na zmianie konfiguracji,
- konfiguracja **końcowa** — to taka konfiguracja, która zawiera pełne rozwiązanie problemu,
- **wynik** — jest to typ reprezentujący rozwiązania; może on być uboższy niż typ konfiguracji; potrzebna jest również operacja **rzutowania** konfiguracji na wynik.
- dla danej konfiguracji określamy **iterator**, który dla danej procedury i konfiguracji, wywołuje tę procedurę dla wszystkich konfiguracji osiągalnych w jednym kroku.

Instancję problemu możemy opisać za pomocą modułów pasujących do następującej sygnatury:

```
module type BT_PROBLEM =
  sig
    (* Typ rozpatrywanych konfiguracji. *)
    type config

    (* Typ szukanych wyników. *)
    type result

    (* Czy dana konfiguracja jest już kompletnym rozwiązaniem. *)
    val final : config ->bool

    (* Wyciąga z konfiguracji istotne rozwiązanie. *)
    (* W przypadku imperatywnych struktur danych tworzy kopię. *)
    val extract : config ->result
```

```

    (* Procedura, która wywołuje daną procedurę, *)
    (* dla każdej konfiguracji osiągalnej w jednym kroku. *)
    val iter : (config ->unit) ->config ->unit
end;;

```

Mechanizm znajdujący jedno rozwiązanie lub wszystkie rozwiązania wygląda następująco:

```

module type BT_SOLVER = functor (Problem : BT_PROBLEM) ->
sig
  (* Wyjątek podnoszony, gdy brak rozwiązań. *)
  exception NoSolution

  (* Procedura znajdująca jedno rozwiązanie. *)
  val onesolution : Problem.config ->Problem.result

  (* Procedura znajdująca wszystkie rozwiązania. *)
  val solutions : Problem.config ->Problem.result list
end;;

(* Implementacja *)
module Bt_Solver : BT_SOLVER = functor (Problem : BT_PROBLEM) ->
struct
  (* Wyjątek podnoszony, gdy brak rozwiązań. *)
  exception NoSolution

  (* Wyjątek podnoszony, gdy znaleziono rozwiązanie. *)
  exception Solution of Problem.result

  (* Procedura znajdująca rozwiązanie. *)
  let onesolution s =
    let rec backtrack s =
      begin
        if Problem.final s then
          raise (Solution (Problem.extract s));
        Problem.iter backtrack s
      end
    in
    try (backtrack s; raise NoSolution)
    with Solution x ->x

  (* Procedura znajdująca wszystkie rozwiązania. *)
  let solutions s =
    let wynik = ref []
    in
    let rec backtrack s =
      begin
        if Problem.final s then
          wynik := (Problem.extract s)::!wynik;
        Problem.iter backtrack s
      end
    in begin

```

```

        backtrack s;
        !wynik
    end
end;;

```

Przedstawione mechanizmy nie uwzględniają problemów optymalizacyjnych ani kryteriów pozwalających obcinać gałęzie przeszukiwania na podstawie znalezionych już rozwiązań. Nie pasuje on również do problemów, w których podjęty wybór sprowadza dany problem do wielu mniejszych problemów.

17.2 Problem ośmiu hetmanów

Jak ustawić n hetmanów na szachownicy $n \times n$ tak, aby żadne dwa się nie atakowały. Jak łatwo zauważyć, w każdym wierszu i każdej kolumnie musi stać dokładnie jeden hetman. Możemy więc starać się je ustawiać w kolejnych kolumnach, sprawdzając, czy kolejny dostawiony hetman nie atakuje już ustawionych. Interesuje nas jeden wynik — pierwszy znaleziony.

Problem ośmiu hetmanów możemy opisać w postaci następującego modułu:

```

module Hetman =
  struct
    (* Pozycje hetmanów na szachownicy. *)
    type result = (int * int) list

    (* Typ konfiguracji na szachownicy. *)
    (* Trójka: wielkość planszy,liczba początkowych kolumn,*)
    (* w których należy postawić hetmany,lista ich pozycji. *)
    type config = int * int * result

    (* Pusta plansza rozmiaru n. *)
    let empty n = (n,n,[])

    (* Pozycje hetmanów w konfiguracji. *)
    let extract (_,_,l) = l

    (* Czy gotowe rozwiązanie. *)
    let final (_,k,_) = k=0

    (* Funkcja określająca,czy dwa hetmany się atakują *)
    let atakuje (x1,y1) (x2,y2) =
      x1=x2 || y1=y2 || x1+y1=x2+y2 || x1-y1=x2-y2

    (* Czy można dostawić hetmana h do ustawionych już hetmanów het *)
    let mozna_dostawic h het =
      fold_left (fun ok x ->ok && not (atakuje h x)) true het

    (* Lista kolejnych liczb całkowitych od-do. *)
    let rec ints a b = if a >b then [] else a :: ints (a+1) b

    (* Konfiguracje powstałe przez dostawienie hetmana w kolumnie k. *)
    let iter p (n,k,l) =
      let r = filter (fun i ->mozna_dostawic (k,i) l) (ints 1 n)

```

```

    in List.iter (fun i -> p (n,k-1,(k,i)::l)) r
end;;

```

Hetmany są ustawiane w kolejnych kolumnach od prawej do lewej. Rozwiązanie jest gotowe, gdy w każdej kolumnie stoi hetman. Kolejne konfiguracje powstają w wyniku ustawienia hetmana w ostatniej pustej kolumnie, we wszystkich miejscach, których nie atakują hetmany stojące już na planszy. Zwróćmy uwagę, że przerywamy sprawdzanie, jak tylko okaże się, że hetmany się atakują.

Dysponując modulem `Hetman`, możemy zastosować do niego funktor `SBt_Solver`.

```

(* Instancja problemu. *)
module H = Bt_Solver (Hetman);;

(* Procedura znajdująca pewne ustawienie hetmanów. *)
let hetman n = H.onesolution (Hetman.empty n);;

(* Procedura znajdująca ustawienie hetmanów. *)
let hetmany n = H.solutions (Hetman.empty n);;

hetman 4;;
- : Hetman.result = [(1,3); (2,1); (3,4); (4,2)]

hetmany 4;;
- : Hetman.result list =
  [[(1,2); (2,4); (3,1); (4,3)];
   [(1,3); (2,1); (3,4); (4,2)]]

hetman 8;;
- : Hetman.result =
  [(1,4); (2,2); (3,7); (4,3); (5,6); (6,8); (7,5); (8,1)]

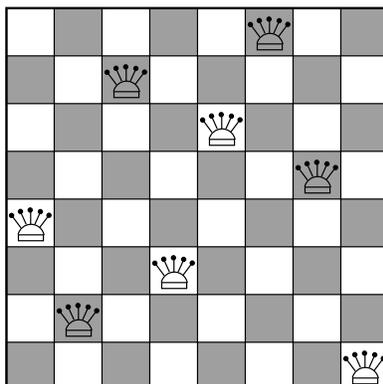
hetmany 8;;
- : Hetman.result list =
  [[(1,5); (2,7); (3,2); (4,6); (5,3); (6,1); (7,4); (8,8)];
   [(1,4); (2,7); (3,5); (4,2); (5,6); (6,1); (7,3); (8,8)];
   [(1,6); (2,4); (3,7); (4,1); (5,3); (6,5); (7,2); (8,8)];
   [(1,6); (2,3); (3,5); (4,7); (5,1); (6,4); (7,2); (8,8)];
   [(1,4); (2,2); (3,8); (4,6); (5,1); (6,3); (7,5); (8,7)];
   [(1,5); (2,3); (3,1); (4,...); ...]; ...]

```

17.3 Przykład: Sudoku

Łamigłówkę Sudoku definiujemy w następujący sposób: Plansza do Sudoku to kwadrat $n \times n$ złożony z mniejszych kwadratów $n \times n$. W rezultacie jest to kwadrat złożony z $n^2 \times n^2$ pól. Sudoku należy wypełnić liczbami od 1 do n^2 , tak żeby w każdej kolumnie, każdym wierszu, a także każdym mniejszym kwadracie każda liczba występowała dokładnie raz. Część liczb jest już na swoich miejscach, pozostałe trzeba odgadnąć.

Sudoku reprezentujemy jako dwuwymiarową tablicę liczb, którą będziemy stopniowo wypełniać. Puste miejsca są wypełnione zerami.



Rysunek 5: Przykładowe ustawienie ośmiu hetmanów na szachownicy.

4					6		5
		7	9	6		1	
	6		4		8		7
	8	3		6			
	1					5	
			2		4	6	
7		6		9		2	
	5		2	4	7		
8		9					6

Rysunek 6: Przykładowe sudoku dla $n = 3$.

```

module Sudoku_Framework =
  struct
    open List

    (* Tablicowa reprezentacja łamigłówki. *)
    (* Tablica  $n^2 \times n^2$  wypełniona liczbami  $0..n^2$ . *)
    (* Puste pola są reprezentowane przez zera. *)
    type sudoku = int array array

    (* Rozmiar n danego sudoku. *)
    let size (s:sudoku) = truncate(sqrt (float (Array.length s)))

    (* Lista kolejnych liczb całkowitych od-do. *)
    let rec ints a b = if a > b then [] else a :: ints (a+1) b

    (* Przekształca parę list w listę par - odpowiednik produktu. *)
    let pairs l1 l2 =
      flatten (map (fun i->map (fun j ->(i,j)) l1) l2)
  end

```

```

(* Lista kolejnych indeksów sudoku rozmiaru n. *)
let indeksy s =
  let n = size s
  in ints 0 (n * n - 1)

(* Lista cyfr,którymi wypełniamy sudoku. *)
let cyfry s =
  let n = size s
  in ints 1 (n * n)

(* Lista par indeksów reprezentujących jeden "kwadrat". *)
let kwadrat s =
  let n = size s
  in
    let s = ints 0 (n-1)
    in pairs s s

(* Tworzy listę indeksów do pustych miejsc w sudoku. *)
let brakujace (s : sudoku) =
  filter
    (fun (i,j) ->s.(i).(j) = 0)
    (pairs (indeksy s) (indeksy s))
  :

```

W dane puste pole możemy wstawić tylko taką liczbę, która nie występuje w danej kolumnie, wierszu i mniejszym kwadracie.

```

:
(* Sprawdza,czy cyfrę k można wstawić w miejscu (i,j). *)
let valid_value (s : sudoku) i j k =
  let valid_column i =
    for_all (fun x ->s.(i).(x) <>k || x = j) (indeksy s)
  and valid_row j =
    for_all (fun x ->s.(x).(j) <>k || x = i) (indeksy s)
  and valid_square i j =
    let n = size s
    in
      for_all
        (fun (x,y) ->(x,y) = (i,j) || s.(x).(y) <>k)
        (map (fun (x,y) ->(n*(i/n) + x,n*(j/n) + y)) (kwadrat s))
    in
      (valid_column i) &&
      (valid_row j) &&
      (valid_square i j)

(* Wyznacza listę cyfr,które można wstawić w miejscu (i,j). *)
let valid_values (s : sudoku) i j =
  filter (valid_value s i j) (cyfry s)
end;;

```

Rozwiązując sudoku, możemy połączyć back-tracking z wnioskowaniem. Jeżeli na podstawie wypełnionych pól potrafimy wypełnić pewne pola, to należy tak zrobić. Jeśli nie potrafimy nic wywnioskować, należy za pomocą back-tracking'u przejrzeć możliwe wartości wybranego pola.

Taką koncepcję możemy zaimplementować za pomocą funktora. Parametrem funktora jest procedura wnioskująca o niektórych polach. Wywnioskowane wartości wstawia ona do tablicy. Jej wynikiem są listy: wypełnionych w wyniku wnioskowania pól oraz pól, które nadal pozostają puste.

```
(* Typ modułu implementującego wnioskowanie n.t. sudoku. *)
module type SUDOKU_REASONING =
  sig
    (* Konfiguracja, to plansza i lista pustych pól. *)
    type config = Sudoku_Framework.sudoku * (int * int) list

    (* Wynikiem reason jest lista pól wywnioskowanych i pozostałych pustych. *)
    val reason : config -> (int * int) list * (int * int) list
  end;;
```

Funktor na podstawie procedury wnioskowania tworzy instancję problemu, który jest rozwiązywany za pomocą generycznego back-tracking'u. Dodatkowo, w back-tracking'u wybieramy do wypełnienia pole, na którym może się znajdować najmniej możliwych liczb.

```
(* Funktor, który wplata procedurę wnioskowania o sudoku *)
(* w back-tracking znajdujący rozwiązania. *)
module Sudoku_BT (R : SUDOKU_REASONING) =
  struct
    open Sudoku_Framework

    (* Instancja problemu dla back-tracking'u. *)
    module Problem =
      struct
        (* Wynikiem jest plansza sudoku. *)
        type result = sudoku

        (* Konfiguracja, to plansza i lista pustych pól. *)
        type config = R.config

        (* Sudoku jest skończone, gdy nie ma wolnych pól. *)
        let final (s,l) = l = []

        (* Kopia planszy. *)
        let copy_sudoku s =
          Array.init (Array.length s) (fun i->Array.copy s.(i))

        (* Extract tworzy kopię planszy. *)
        let extract (s,l) = copy_sudoku s

        (* Procedura przeglądająca konfiguracje osiągalne w jednym kroku. *)
        (* Wypełnianie jest pierwsze pole z listy pustych pól. *)
      end
  end
```

```

let iter p (s,l) =
  if not (final (s,l)) then
    let (wst,l1) = R.reason (s,l)
    in begin
      if not (final (s,l1)) then begin
        (* Wybierz z listy pozycji do wypełnienia tę o najmniejszej *)
        (* liczbie możliwych miejsc do wstawienia. *)

        (* Lista pozycji i możliwych ruchów, posortowana. *)
        let lr =
          let cmp (x,_,_) (y,_,_) =
            if x < y then -1 else if x > y then 1 else 0
          in
            List.sort cmp
              (List.map
                (fun (i,j) ->
                  let v = valid_values s i j
                  in (List.length v, (i,j), v))
                l1)
          in
            (* Pole z najkrotsza listą możliwości. *)
            let (_, (i,j), v) = List.hd lr
            (* Pozostałe pola. *)
            and t = List.map (fun (_,p,_) -> p) (List.tl lr)
            in
              List.iter
                (fun k ->
                  begin
                    s.(i).(j) <- k;
                    p (s,t);
                    s.(i).(j) <- 0
                  end)
                v
            end else p (s,l1);
            List.iter (fun (i,j) -> s.(i).(j) <- 0) wst
          end
        end
      end
    end

  (* Instancja rozwiązania za pomocą back-tracking. *)
module Solution = Bt_Solver (Problem)

(* Procedura znajdująca rozwiązania dla danej planszy sudoku. *)
let sudoku s =
  Solution.solutions (s, brakujace s)

end;;

```

To, że w pierwszej kolejności wypełniamy pole, dla którego liczba możliwych cyfr jest minimalna, realizuje dwie dodatkowe strategie optymalizacyjne. Po pierwsze, jeżeli na planszy jest pole, na którym nie może znajdować się żadna cyfra, czyli sudoku nie da się rozwiązać, ucinamy wszelkie wywołania rekurencyjne dla tej konfiguracji. Jeżeli na planszy znajduje się

pole, na którym może znajdować się tylko jedna cyfra, to w pierwszej kolejności wstawiana jest ta właśnie cyfra.

Prosty back-tracking możemy uzyskać nie stosując żadnego wnioskowania:

```
(* Brak wnioskowania o sudoku. *)
module NoReasoning =
  struct
    (* Konfiguracja, to plansza i lista pustych pól. *)
    type config = Sudoku_Framework.sudoku * (int * int) list

    (* Nic nie wnioskujemy. *)
    let reason (s,l) = ([],l)
  end;;
```

```
module Sudoku1 = Sudoku_BT (NoReasoning);;
```

Nie jest to jednak zbyt efektywny algorytm, choć wystarczający do rozwiązywania prostych sudoku dla $n = 3$.

```
module Reasoning =
  struct
    open Sudoku_Framework

    (* Konfiguracja, to plansza i lista pustych pól. *)
    type config = sudoku * (int * int) list

    (* Sprawdza czy dana wartość występuje na liście. *)
    let present x l = List.filter (fun y ->x = y) l <>[]

    (* Jeżeli w wierszu jest tylko jedno miejsce, w którym może *)
    (* występować jakaś cyfra, to jest tam ona wstawiana. *)
    let reason_row (s,l) =
      let changed = ref true
        and puste = ref l
        and wstawione = ref []
        and n = size s
      in
        while !changed do
          changed := false;
          (* Kolejne wiersze. *)
          for i = 0 to n * n - 1 do
            (* Wartości, które mogą pojawić się na poszczególnych miejscach. *)
            let v =
              Array.init (n*n)
                (fun j ->if s.(i).(j) = 0 then valid_values s i j else [s.(i).(j)])
            in
              (* Przejrzyj kolejne cyfry. *)
              for k = 1 to n * n do
                (* Sprawdź, na ilu pozycjach może występować cyfra. *)
                let jj = List.filter (fun j ->present k v.(j)) (indeksy s)
                in
```

```

        (* Jeśli pozycja cyfry jest wyznaczona *)
        (* jednoznacznie, wypełnij ją. *)
        match jj with
        [j] -> if s.(i).(j) = 0 then begin
            changed := true;
            s.(i).(j) <- k;
            wstawione := (i,j)::!wstawione
        end |
        _ -> ()
        done
    done;
    (* Korekta listy pustych pól. *)
    puste := List.filter (fun (i,j) -> s.(i).(j)=0) (!puste);
done;
(!wstawione,!puste)

(* Jeżeli w kolumnie jest tylko jedno miejsce, w którym może *)
(* występować jakaś cyfra, to jest tam ona wstawiana. *)
let reason_col (s,l) =
    let changed = ref true
    and puste = ref l
    and wstawione = ref []
    and n = size s
    in
        while !changed do
            :
        done;
        (!wstawione,!puste)

(* Jeżeli w małym kwadracie jest tylko jedno miejsce, w którym *)
(* może występować jakaś cyfra, to jest tam ona wstawiana. *)
let reason (s,l) =
    :

end;;

module Sudoku2 = Sudoku_BT (Reasoning);;

```

17.4 Optymalizacje

Zasadnicze znaczenie dla złożoności back-trackingowego ma obcinanie gałęzi nieprowadzących do interesujących nas wyników oraz zmniejszenie liczby rozgałęzień rekurencyjnych. Back-tracking ma zwykle koszt wykładniczy. Zmniejszenie liczby rozgałęzień rekurencyjnych odpowiada zmniejszeniu podstawy potęgi wyrażającej koszt czasowy.

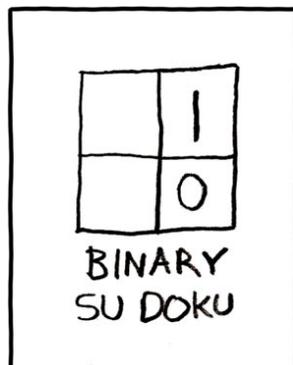
Możemy starać się obcinać gałęzie, stosując proste kryteria negatywne. Np. proste warunki, z których wynika, że skonstruowanej części rozwiązania nie da się uzupełnić do poprawnego rozwiązania. Jeżeli szukamy rozwiązania pod jakimś względem optymalnego, to możemy starać się oszacować od dołu koszt rozwiązań powstałych z rozwinięcia częściowego rozwiązania. Jeżeli koszt ten jest na pewno większy od kosztu znalezionej już rozwiązania, to nie warto przeglądać takich gałęzi.

Warto również zwrócić uwagę na kolejność rozpatrywania kolejnych wariantów. Należy najpierw wybierać takie, które mają szansę dać lepsze oszacowania lub lepiej rokują na rozwiązanie. Wówczas mamy szansę wcześniej znaleźć rozwiązanie lub obciąć więcej gałęzi.

[[Przykład na obcinanie na podstawie już uzyskanych wyników]]

[[Rozwiązanie początkowe i końcowe są dane – szukamy konfiguracji w środku.]]

17.5 Deser



<http://xkcd.com/74/>

Ćwiczenia

1. Problem obejścia szachownicy skoczkiem.
2. Ustawić minimalną liczbę hetmanów atakujących wszystkie pozostałe pola na szachownicy,
3. Napisz procedurę **pierwsze**, która dla danej liczby całkowitej n ($n > 1$) wyznacza rozbić n na sumę minimalnej liczby składników będących liczbami pierwszymi. (Możesz założyć prawdziwość hipotezy Goldbacha.)
4. Dany jest (multi-)zbiór kostek domina. Należy wyznaczyć maksymalny łańcuch, jaki można ułożyć z danych kostek, oczywiście zgodnie z zasadami domina. Rozwiązanie powinno mieć postać procedury **domino** : $(\alpha \times \alpha) \text{ list} \rightarrow (\alpha \times \alpha) \text{ list}$. Klocki można obracać.
5. [III OI] Mokra robota. Dany jest zestaw naczyń szklanych o określonych objętościach. Dla każdego z nich wiemy ile chcemy, aby było do niego nalane wody. Początkowo wszystkie naczynia są napełnione wodą. Dozwolone operacje to:
 - przelanie całej zawartości jednego naczynia do drugiego, pod warunkiem, że się zmieści,
 - dolanie wody do pełna z jednego naczynia do drugiego,
 - wylanie wody z naczynia do zlewu.

Szukamy takiej sekwencji ruchów, po wykonaniu której we wszystkich naczyniach jest tyle wody, ile chcieliśmy. (Niekoniecznie back-tracking, ale przeszukiwanie przestrzeni stanów.)

6. [XIV OI] Atrakcje turystyczne (uproszczone). Mamy n miejscowości, ponumerowanych od 1 do n . Odległości między miejscowościami są dane w postaci (symetrycznej) tablicy liczb całkowitych, o wymiarach $n \times n$.

Chcemy zwiedzić wszystkie te miejscowości, jednak nie w dowolnej kolejności. Mamy daną listę ograniczeń postaci $[(i_1, j_1); \dots; (i_k, j_k)]$. Ograniczenie (i, j) oznacza, że miasto i musi być zwiedzone przed miastem j . Możesz założyć, że ograniczenia da się spełnić.

Na początku wyruszamy z miasta 1, a kończymy podróż w mieście n . Wyznacz minimalną drogę, jaką trzeba przejechać, żeby zwiedzić wszystkie miasta.

Wykład 18. Technika spamiętywania

18.1 Technika spamiętywania

Mając do dyspozycji mapy, możemy zapisać rozwiązanie w sposób rekurencyjny, unikając jednak wielokrotnych wywołań rekurencyjnych dla tych samych wartości argumentów. Korzystamy w tym celu z techniki *spamiętywania*. Technika ta polega na pamiętaniu wszystkich obliczonych uprzednio rozwiązań podproblemów. Jest to skrzyżowanie rozwiązania rekurencyjnego i programowania dynamicznego, o którym powiemy dokładniej w kolejnym wykładzie.

Zobaczmy, jak wygląda zastosowanie opisanej techniki do obliczania liczb Fibonacciego:

```
let tab = ref empty;;
let rec fib n =
  if dom !tab n then
    apply !tab n
  else
    let wynik = if n <2 then n else fib (n-1) + fib (n-2)
    in begin
      tab := update !tab n wynik;
      wynik
    end;;
```

Schemat spamiętywania jest dosyć uniwersalny: w implementacji rekurencyjnej, ilekroć chcemy wywołać rekurencyjnie procedurę rozwiązującą podproblem, sprawdzamy, czy już takie wywołanie nie miało miejsca, a jeżeli tak, to pobieramy obliczony wcześniej wynik. Możemy zaimplementować coś w rodzaju „otoczki”, która filtruje wywołania procedur. Mechanizm ten możemy wyłuskać i przedstawić w postaci procedury wyższego rzędu. Parametrami tej procedury są: tablica użyta do spamiętywania, spamiętywana funkcja i jej argument.

```
let memoize tab f x =
  if dom !tab x then
    apply !tab x
  else
    let wynik = f x
    in begin
      tab := update !tab x wynik;
      wynik
    end;;
```

Ponownie, procedurę obliczającą liczby Fibonacciego możemy zapisać następująco:

```
let fib =
  let tab = ref empty
  in
    let rec f n =
      memoize tab (function n ->
        if n <2 then n else f (n-1) + f (n-2)) n
    in f;;
```

18.2 Problem NWP i algorytm

- Problem najdłuższego wspólnego podciągu.
- Uogólnienie problemu na prefiksy danych ciągów.
- Własność podproblemu, zależności między rozwiązaniami podproblemów:
 - Jeśli $x_i = y_j$, to $nwp(i, j) = nwp(i - 1, j - 1) @ [x_i]$.
 - Jeśli ostatni wyraz nwp jest różny od x_i , to $nwp(i, j) = nwp(i - 1, j)$.
 - Jeśli ostatni wyraz nwp jest różny od y_j , to $nwp(i, j) = nwp(i, j - 1)$.
 - Funkcja nwp jest monotoniczna względem obydwu współrzędnych.
 - Jeśli $x_i \neq y_j$, to $nwp(i, j)$ jest dłuższym z podciągów $nwp(i - 1, j)$ i $nwp(i, j - 1)$.
 - Warunki brzegowe: $nwp(i, 0) = nwp(0, j) = []$.
- Algorytm — tabela rozwiązań dla podproblemów i jej wypełnianie.

	A	B	C	B	D	A	B
	0	0	0				
B	0	0	1	1	1		
D	0	0	1	1	1	2	
C	0	0	1	2	2	2	
A		1	1	2	2	2	3
B					3	3	4
A						4	4

Wszystkie najdłuższe wspólne podciągi to:

- BCBA,
- BCAB,
- BDAB,

przy czym ostatni z nich występuje w pierwszym ciągu na dwa sposoby.

18.3 Najdłuższy wspólny podciąg — implementacja ze spamiętywaniem

W poniższym programie mapa przyporządkowuje parom liczb (i, j) pary (najdłuższy wspólny podciąg, jego długość). Wynikiem procedury `pom` jest para: (odwrócony) najdłuższy wspólny podciąg i jego długość. Jako efekt uboczny obliczeń powstaje również mapa zawierająca wszystkie obliczone wyniki podproblemów. Wynikiem całej procedury jest najdłuższy wspólny podciąg dwóch ciągów.

```
let nwp ciag_x ciag_y =
  let x = Array.of_list ciag_x
  and y = Array.of_list ciag_y
```

```

and tab = ref empty
in
  let rec pom (i,j) =
    memoize tab (fun (i,j) ->
      if i = 0 || j = 0 then ([],0)
      else if x.(i-1) = y.(j-1) then
        let (c,l) = pom (i-1,j-1)
        in (x.(i-1)::c,l+1)
      else
        let (c1,l1) = pom (i-1,j)
        and (c2,l2) = pom (i,j-1)
        in
          if l1 > l2 then (c1,l1) else (c2,l2)
    ) (i,j)
  in
    rev (fst (pom (length ciag_x,length ciag_y)));

nwp ['A'; 'B'; 'C'; 'B'; 'D'; 'A'; 'B']
    ['B'; 'D'; 'C'; 'A'; 'B'; 'A'];;
- : char list = ['B'; 'D'; 'A'; 'B']

```

Ćwiczenia

1. Jaka jest minimalna liczba monet lub banknotów potrzebna do wydania n zł reszty, przyjmując, że w obrocie są dostępne monety o zadanych (w postaci listy) nominałach?
2. Jaka jest minimalna liczba monet lub banknotów potrzebna do wydania n zł reszty, przyjmując, że monety mogą być przekazywane w obie strony. (Na przykład, żeby otrzymać 9 zł mogą dać 1 zł i otrzymać 10 zł, razem 2). W obrocie są dostępne monety i banknoty o zadanych (w postaci tablicy) nominałach.
3. Na ile sposobów można wydać n zł reszty przyjmując, że w obrocie są dostępne monety o zadanych (w postaci listy) nominałach.
4. Dana jest kwota k oraz zestaw nominałów banknotów $b = [b_1; b_2; \dots; b_n]$. Mamy dowolną liczbę banknotów każdego z nominałów. Napisz procedurę `banknoty : int → int list → int`, która dla danych k i b obliczy minimalną liczbę różnych nominałów banknotów, jakich należy użyć do wydania kwoty k . Jeżeli nie da się wydać kwoty k , używając podanych nominałów, to procedura powinna zwrócić -1 .
Na przykład, `banknoty 49 [10, 15, 11, 12, 20] = 3`. Liczbę 49 można uzyskać na dwa sposoby: $49 = 12 + 12 + 10 + 15 = 11 + 11 + 15 + 12$.
5. [XII OI, Banknoty] W bankomacie znajdują się banknoty o nominałach b_1, b_2, \dots, b_n , przy czym banknotów o nominale b_i jest w bankomacie c_i . Jak wypłacić zadaną kwotę używając jak najmniejszej liczby banknotów?
6. Napisz procedurę, która dla danego ciągu liczb wyznaczy jego najdłuższy rosnący podciąg.
7. Napisz procedurę `pierwsze`, która dla danej liczby całkowitej n ($n > 1$) wyznacza rozbięcie n na sumę minimalnej liczby składników będących liczbami pierwszymi. (Tym razem nie zakładaj prawdziwości hipotezy Goldbacha.)
8. Dana jest liniowa plansza z polami ponumerowanymi od 1 do n , obdarzonymi rzeczywistymi wagami a_1, a_2, \dots, a_n . Na polu 1 stoi pionek, który może poruszać się po planszy skokami wprzód lub w tył na odległość nieprzekraczającą k , aż zatrzyma się definitywnie na polu n (pionek może odwiedzić pola 1 i n więcej niż raz). Każde pole, na którym stanie pionek jest zbite.
Napisać funkcję `chocki`, która dla danego k oraz listy $(a_1 a_2 \dots a_n)$ zwróci maksymalną możliwą sumę wag pól niezbitych.

Wykład 19. Programowanie dynamiczne

19.1 Implementacja imperatywna, z wykorzystaniem tablic

W przypadku problemu najdłuższego wspólnego podciągu, możemy spamiętywanie zaimplementować wprost, gdyż najistotniejszym wynikiem jest tablica, która powstaje w trakcie obliczeń.

```
let nwp ciag_x ciag_y =
  if (ciag_x = []) || (ciag_y = []) then [] else
  let n = length ciag_x
  and m = length ciag_y
  and x = Array.of_list ((hd ciag_x)::ciag_x)
  and y = Array.of_list ((hd ciag_y)::ciag_y)
  in
  (* a.(i).(j) = length (nwp [x_1;...;x_i] [y_1;...;y_j]) *)
  let a = Array.make_matrix (n+1) (m+1) 0
  in
  (* Rekonstrukcja nwp na podstawie tablicy długości. *)
  let rec rekonstrukcja acc i j =
    if i = 0 || j = 0 then acc
    else if x.(i) = y.(j) then
      rekonstrukcja (x.(i)::acc) (i-1) (j-1)
    else if a.(i).(j) = a.(i-1).(j) then
      rekonstrukcja acc (i-1) j
    else
      rekonstrukcja acc i (j-1)
  in
  begin
    (* Wypełnienie tablicy a *)
    for i = 1 to n do
      for j = 1 to m do
        if x.(i) = y.(j) then
          (* Odpowiadające sobie elementy ciągów są równe. *)
          a.(i).(j) <- a.(i-1).(j-1) + 1
        else
          (* Wybór wariantu realizującego dłuższy podciąg. *)
          a.(i).(j) <- max a.(i-1).(j) a.(i).(j-1)
        done
      done;
    (* Rekonstrukcja wyniku. *)
    rekonstrukcja [] n m
  end;;

nwp ['A'; 'B'; 'C'; 'B'; 'D'; 'A'; 'B'] ['B'; 'D'; 'C'; 'A'; 'B'; 'A'];;
```

19.2 Implementacja NWP oparta o mapy

- Główna procedura:

```

let nwp ciag_x ciag_y =
  let
    a = zbuduj_tablice1 ciag_x ciag_y
  in
    odtworz_podciag1 a ciag_x ciag_y;;

```

- Budowa tablicy:

```

let dlugosc u v w x y =
  if x = y then
    v + 1
  else
    max u w;;

```

```

let zbuduj_tablice ciag_x ciag_y =

```

```

  (*
    Dodaje kolejny wiersz tablicy,
    y= rozpatrywany element ciągu y,j = nr wiersza. *)
  let dodaj_wiersz tablica y j =
    let (t,_) =
      fold_left
        (fun (tab,i) x ->
          let u = apply tab (j,i-1)
            and v = apply tab (j-1,i-1)
            and w = apply tab (j-1,i)
          in
            (update tab (j,i) (dlugosc u v w x y),i+1))
        (update tablica (j,0) 0,1)
      ciag_x
    in t
  in
    (* Pierwszy wiersz tablicy - same zera *)
    let pierwszy =
      let (a,_) =
        fold_left
          (fun (a,i) _ ->(update a (0,i) 0,i+1))
          (update empty (0,0) 0,1)

```

```

        ciag_x
    in a

(* Budowanie tablicy,
   yl= pozostale elementy ciągu y,j = nr wiersza. *)
and buduj a yl =
    let (w,_) =
        fold_left (fun (a,j) y ->(dodaj_wiersz a y j,j+1)) (a,1) yl
    in w
in
    buduj pierwszy ciag_y;;

```

- Rekonstrukcja podciągu:

```

(* Rekonstrukcja ciągu na podstawie tablicy *)
let odtworz_podciag tablica ciag_x ciag_y =
    let rec odtworz akumulator tab ciag_x ciag_y i j =
        match ciag_x with
        [] ->akumulator |
        (x::tx) ->
            match ciag_y with
            [] ->akumulator |
            (y::ty) ->
                if x = y then
                    odtworz (x::akumulator) tab tx ty (i-1) (j-1)
                else
                    let u = apply tab (j,i-1)
                    and w = apply tab (j-1,i)
                    in
                        if u >w then
                            odtworz akumulator tab tx ciag_y (i-1) j
                        else
                            odtworz akumulator tab ciag_x ty i (j-1)
    in
        odtworz [] tablica
        (rev ciag_x) (rev ciag_y)
        (length ciag_x) (length ciag_y);;

```

19.3 Efektywna funkcyjna implementacja NWP, efektywna, acz skomplikowana

- Struktura tablicy

```
type array = { value : int; up : array; upleft : array; left : array };;  
let rec zero = { value = 0; up = zero; upleft = zero; left = zero };;
```

- Wypełnianie tablicy

```
(* A[i,j] = długość najdłuższego wspólnego podciągu *)  
(* (x_1,...,x_i) i (y_1,...,y_j). *)  
(* Tablica jest reprezentowana w postaci odwróconej listy *)  
(* odwróconych list i uzupełniona strażnikami równymi 0. *)  
let zbuduj_tablice ciag_x ciag_y =  
  (* Dodaje kolejny wiersz tablicy,  
   y= rozpatrywany element ciągu y.  
  *)  
  let dodaj_wiersz tab y =  
    let rec dodawaj tab_up lista =  
      match lista with  
      [] ->zero |  
      (x::t) ->  
        let tab_upleft = tab_up.left  
        in  
        let tab_left = dodawaj tab_upleft t  
        in  
        let v = dlugosc tab_left.value  
                tab_upleft.value  
                tab_up.value  
                x y  
        in  
        { value = v;  
          left = tab_left;  
          upleft = tab_upleft;  
          up = tab_up  
        }  
    in  
    dodawaj tab (rev ciag_x)  
  in  
  fold_left dodaj_wiersz zero ciag_y;;
```

- Odtworzenie ciągu na podstawie tablicy długości

```

let odtworz_podciag tablica ciag_x ciag_y =
  let rec odtworz akumulator tab ciag_x ciag_y =
    match ciag_x with
    [] ->akumulator |
    (x::tx) ->
      match ciag_y with
      [] ->akumulator |
      (y::ty) ->
        if x = y then
          odtworz (x::akumulator) tab.upleft tx ty
        else
          let lv = tab.left.value
          and uv = tab.up.value
          in
            if lv > uv then
              odtworz akumulator tab.left tx ciag_y
            else
              odtworz akumulator tab.up ciag_x ty
  in
    odtworz [] tablica (rev ciag_x) (rev ciag_y);;

```

19.4 Ogólny schemat programowania dynamicznego

- Sparametryzowanie problemu — zamiast jednego problemu, rozwiązujemy całą gamę powiązanych podproblemów.
- Wyznaczenie zależności między rozwiązaniami podproblemów.
- Rodzaj struktury danych, w której przechowujemy rozwiązania podproblemów.
- Kolejność rozwiązywania podproblemów.
- Czy można lepiej sformułować podproblemy?
- Ewentualne wyznaczenie pomocniczych problemów, których wcześniejsze obliczenie pozwala zmniejszyć złożoność czasową.
- Ewentualna rekonstrukcja wyniku na podstawie obliczonych wyników częściowych.
- Polepszenie złożoności pamięciowej.
- Czy lepiej zastosować spamiętywanie, czy programowanie dynamiczne?

Ćwiczenia

1. Antyczny bardzo długi kijek został połamany na kawałki. Należy go skleić. Wszystkie kawałki zostały już ułożone w odpowiedniej kolejności, ale nie wiadomo ile potrzeba kleju. Długości kolejnych kawałków są dane w postaci listy $l = [x_1; x_2; \dots; x_n]$. Do sklejenia kawałków długości a i b potrzeba $\max(a, b)$ ml kleju. Po ich sklejeniu otrzymujemy jeden kawałek długości $a + b$.

Napisz procedurę `klej : int list → int`, która na podstawie długości kawałków obliczy minimalną ilość kleju (w ml) potrzebną do sklejenia wszystkich kawałków.

2. Dana jest dwuwymiarowa kwadratowa szachownica o rozmiarach $n \times n$ wypełniona dodatnimi liczbami całkowitymi, po której będziemy się poruszać startując z lewego górnego rogu (komórka o indeksach $(0).(0)$). Dozwolone posunięcia to:

- przejście na sąsiednie lewe pole — zmniejszenie drugiego indeksu,
- przejście na sąsiednie prawe pole — zwiększenie drugiego indeksu,
- przejście na sąsiednie dolne pole — zwiększenie pierwszego indeksu.

Celem jest dojście do prawego dolnego rogu (komórka o indeksach $(n-1).(n-1)$) tak, aby suma liczb na odwiedzonych polach była jak najmniejsza.

Napisz procedurę `przejscie : int array array → int`, która dla danej szachownicy oblicza najmniejszą możliwą sumę pól odwiedzonych w trakcie takiego przejścia.

3. [XII OI, zadanie *Autobus*, uproszczone] Ulice miasta tworzą szachownicę — prowadzą z północy na południe lub ze wschodu na zachód, a każda ulica prowadzi na przestrzał przez całe miasto — każda ulica biegnąca z północy na południe krzyżuje się z każdą ulicą biegnącą ze wschodu na zachód i vice versa. Ulice prowadzące z północy na południe są ponumerowane od 1 do n , w kolejności z zachodu na wschód. Ulice prowadzące ze wschodu na zachód są ponumerowane od 1 do m , w kolejności z południa na północ. Każde skrzyżowanie i -tej ulicy biegnącej z północy na południe i j -tej ulicy biegnącej ze wschodu na zachód oznaczamy parą liczb (i, j) (dla $1 \leq i \leq n$, $1 \leq j \leq m$).

Po ulicach miasta kursuje autobus. Zaczyna on trasę przy skrzyżowaniu $(1, 1)$, a kończy przy skrzyżowaniu (n, m) . Ponadto autobus może jechać ulicami tylko w kierunku wschodnim i/lub północnym.

Przy niektórych skrzyżowaniach znajdują się pasażerowie oczekujący na autobus. Rozmieszczenie pasażerów jest dane w postaci prostokątnej tablicy a $n \times m$ liczb całkowitych — $a.(i).(j)$ oznacza liczbę pasażerów czekających przy skrzyżowaniu (i, j) .

Napisz procedurę `autobus`, która na podstawie tablicy a obliczy maksymalną liczbę pasażerów, których może zabrać autobus. Zakładamy, że w autobusie zmieści się dowolna liczba pasażerów.

4. [CLR, ćw. 16-4] Firma planuje zorganizować przyjęcie. Hierarchia stanowisk w firmie ma strukturę drzewa (ogólnego). Każdy pracownik charakteryzuje się pewną „towarzystwością” wyrażoną liczbą dodatnią. Napisz program, który dobierze gości tak, aby:

- na przyjęciu nie był obecny bezpośredni przełożony żadnego z gości,
- suma współczynników towarzyskości gości była maksymalna.

Jak zapewnić, aby szef firmy był na przyjęciu?

```
type tree = Node of (int * tree list);;
let rec impreza (Node (wsp,sons)) =
  let lp = map impreza sons
  in
  fold_left (fun (z,bez) (x,y) ->(z + y,bez + max x y))
    (wsp,0) lp;;
```

5. Dana jest deklaracja typu:

```
type 'a drzewo =
  Node of 'a * 'a drzewo * 'a drzewo |
  Null
```

Napisz procedurę `środek : 'a drzewo -> 'a drzewo`, która znajduje w zadanym drzewie taki węzeł (Node), dla którego maksymalna spośród jego odległości od liści jest jak najmniejsza. Wynikiem tej procedury powinno być poddrzewo zakorzenione w wyznaczonym węźle.

Co się zmieni, jeżeli będziemy chcieli znaleźć wierzchołek, dla którego minimalna spośród jego odległości do liści jest jak największa?

6. Na rozgałęzieniach drzewa siedzą wróble. Każde rozgałęzienie drzewa waży 1 kilogram, a nieujemna waga siedzących na tym rozgałęzieniu wróbli podana jest w strukturze drzewiastej:

```
type drzewo = Null | Node of int * drzewo * drzewo
```

Waga drzewa to suma wag rozgałęzień i siedzących na nich wróbli. Drzewo nazwiemy zrównoważonym, jeśli w każdym rozgałęzieniu waga dwóch poddrzew będzie taka sama. Rzucając celnie kamieniem w rozgałęzienie drzewa możemy przegonić wszystkie wróble z poddrzewa zaczepionego w tym rozgałęzieniu. Napisz funkcję `rownowaga : drzewo -> bool`, która dla danego drzewa określa, czy da się tak rzucać kamieniami w drzewo, aby stało się ono zrównoważone. Podaj złożoność czasową i pamięciową swojego rozwiązania.

Dla przykładu, aby zrównoważyć następujące drzewo:

```
Node (3,Node (5,Node (1,Null,Null),Node (7,Null,Null)),Node (2,Null,Null))
```

wystarczy raz rzucić kamieniem (w jego lewe poddrzewo).

7. Każda latarnia jest opisana przez trójkę postaci (x, y, p) oznaczającą, że latarnia może oświetlić (domknięty) odcinek alejki od x do y i pobiera wtedy p W energii elektrycznej ($x < y, p > 0$). Dana jest lista trójek opisujących latarnie, oraz para liczb (a, b) , $a < b$, opisująca odcinek alejki, który chcemy oświetlić.

Napisz procedurę `moc : (int * int * int) list → (int * int) → int`, która obliczy minimalną moc potrzebną do oświetlenia danego odcinka alejki. Jeżeli jego oświetlenie nie jest możliwe, to poprawnym wynikiem jest -1 .

Przykład:

```
moc [(2, 5, 10), (10, 12, 4), (-1, 2, 8), (6, 9, 10), (9, 15, 20),  
      (4, 7, 8), (0, 12, 100), (8, 10, 4), (-2, 2, 6)] (1, 11) = 42
```

8. [XVI OI, zadanie *Konduktor*] Pociąg na trasie zatrzymuje się na n stacjach, ponumerowanych od 0 do $n-1$. Dana jest tablica a , taka że $a.(i).(j)$ (dla $0 \leq i < j < n$) jest równe liczbie pasażerów, którzy wsiadają na stacji i i wysiadają na stacji j . Możesz wybrać k momentów (między kolejnymi stacjami), kiedy konduktor sprawdza bilety. Oblicz, jaka jest maksymalna liczba pasażerów, którym zostaną sprawdzone bilety.
9. Mamy dany (w postaci listy) ciąg dodatnich liczb całkowitych $[x_1; x_2; \dots; x_n]$. Na ciągu tym można wykonywać operacje „sklejania”, tzn. dwa kolejne elementy ciągu można zastąpić jednym, równym ich sumie. Napisz procedurę:

`sklejanie : int list → int`

obliczającą minimalną liczbę operacji „sklejania”, które należy wykonać, aby ciąg był rosnący. Na przykład, `sklejanie [3; 5; 4; 2; 7] = 1`.

Równoważne zadanie pojawiło się na konkursie USACO Open Gold w 2009 r. i zostało opisane w *Delcie* z maja 2016 r.:

http://www.deltami.edu.pl/temat/informatyka/algorytmy/2016/04/28/Wieza_z_siana/.

10. Dana jest dwuwymiarowa tablica a wypełniona nieujemnymi liczbami całkowitymi. Reprezentuje ona mapę prostokątnego lasu, a wartości tablicy odpowiadają liczbie drzew rosnących w różnych miejscach. Napisz procedurę `prostokt : int array → int → int`, która dla zadanej tablicy a oraz liczby k znajdzie w obrębie tablicy (lasu) taki prostokąt, który:

- zawiera przynajmniej k drzew, tzn. suma elementów tablicy w prostokącie jest $\geq k$,
- obwód prostokąta jest minimalny.

Wynikiem procedury powinien być obwód tego prostokąta. Podaj złożoność czasową i pamięciową swojego rozwiązania.

Prostokąt o dolnym lewym rogu (x_1, y_1) i górnym prawym (x_2, y_2) zawiera wszystkie elementy tablicy $a.(i).(j)$ dla $x_1 \leq i \leq x_2, y_1 \leq j \leq y_2$, oraz ma obwód $2 \cdot (x_2 - x_1 + 1) + 2 \cdot (y_2 - y_1 + 1)$.

11. Dana jest prostokątna mapa złóż gazu. Mapa jest dana w postaci dwuwymiarowej prostokątnej tablicy nieujemnych liczb całkowitych, oznaczających ilość znajdującego się w danym miejscu gazu.

Chcemy zakupić prawa do wydobywania gazu z terenu, który ma kształt prostokąta i zawiera łącznie przynajmniej k jednostek gazu. Napisz procedurę `gaz : int array array → int → int`, która obliczy minimalną powierzchnię takiego terenu.

12. Mamy most linowy długości N metrów ($N \geq 1$). Dla każdego metrowego odcinka mostu znamy jego wytrzymałość, to jest maksymalny ciężar, jaki może znaleźć się łącznie na moście nie powodując rozerwania lin na tym odcinku. Możemy rozmieścić K podpór, dzieląc efektywnie most na $K+1$ przylegających do siebie połączonych mostów. Podpory mają znikomą szerokość i rozmieszczamy je na styku metrowych odcinków mostu.

Przed wjazdem na most zostaną ustawione dwa ograniczenia: maksymalny dopuszczalny ciężar pojazdu C jaki może wjechać na most, oraz minimalna odległość D jaką należy zachować między pojazdami. Jeśli dwie kolejne podpory znajdują się w odległości X , to odcinki łączącego je mostu muszą mieć wytrzymałość przynajmniej $\lceil \frac{X}{D} \rceil \cdot C$.

Ograniczenie C będzie równe minimalnej wytrzymałości całego mostu. Twoim zadaniem jest wyznaczenie minimalnej możliwej wartości ograniczenia D , jakie można uzyskać odpowiednio rozmieszczając podpory. Napisz procedurę `most : int array → int → int`, która mając dane wytrzymałości odcinków mostu oraz liczbę podpór K wyznaczy minimalną możliwą wartość ograniczenia D .

13. [XV OI, zadanie *Kupno gruntu*] Dana jest kwadratowa tablica $n \times n$ nieujemnych liczb całkowitych a , oraz dodatnia liczba całkowita k . Napisz procedurę `prostokat : int array array → int → (int * int) * (int * int)`, która dla tablicy a i liczby k znajdzie taki prostokątny obszar w obrębie tej tablicy, że suma liczb z tego obszaru jest między k , a $2k$. Ścisłe mówiąc, jeśli `prostokat a k = ((x1, y1), (x2, y2))`, to $k \leq \sum_{i=x_1}^{x_2} \sum_{j=y_1}^{y_2} a.(i).(j) \leq 2k$. Możesz założyć, że taki obszar istnieje.

14. Dana jest deklaracja typu drzew binarnych:

```
type  $\alpha$  drzewo = Puste | Węzeł of  $\alpha$  *  $\alpha$  drzewo *  $\alpha$  drzewo;;
```

Dla każdego węzła w drzewie definiujemy jego *odchylenie*: patrzymy na ścieżkę prowadzącą od korzenia do danego węzła, odchylenie to różnica między liczbą krawędzi idących w prawo a liczbą krawędzi idących w lewo na tej ścieżce. Jeżeli krawędzi idących w lewo jest więcej, to odchylenie jest liczbą ujemną. *Szerokością* drzewa nazwiemy różnicę między maksymalnym i minimalnym odchyleniem jego węzłów. (Przyjmujemy, że puste drzewo ma szerokość 0.)

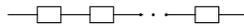
Napisz procedurę `szerokosc : α drzewo → int` obliczającą szerokość drzewa.

15. [X OI, zadanie *Płytki drukowane*] Firma Bajtel rozpoczyna produkcję elektronicznych układów szeregowo-równoległych. Każdy taki układ składa się z części elektronicznych, połączeń między nimi oraz dwóch połączeń doprowadzających prąd. Układ szeregowo-równoległy może składać się z:

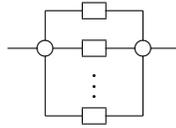
- pojedynczej części,



- kilku mniejszych układów szeregowo-równoległych połączonych szeregowo,



- dwóch części rozgałęziających łączących równolegle kilka mniejszych układów szeregowo-równoległych.



Układy są montowane na dwustronnych płytkach drukowanych. Problem polega na ustaleniu, które połączenia powinny znaleźć się na górnej, a które na dolnej stronie płytki. Ze względów technicznych, jak najwięcej połączeń powinno znaleźć się na dolnej stronie płytki, jednak do każdej części musi dochodzić przynajmniej jedno połączenie znajdujące się na górnej stronie płytki.

Struktura układu szeregowo-równoległego jest opisana za pomocą wartości następującego typu:

```
type układ =
  Element |
  Szeregowo of układ list |
  Równoległe of układ list;;
```

Napisz procedurę `pytki : układ → int`, która dla danego układu wyznaczy minimalną liczbę połączeń, które muszą być umieszczone na górnej stronie płytki.

- [BOI'2003, przeformułowane] Tomcio chce zrobić model drzewa (acyklicznego nieskierowanego grafu spójnego) z nitki i szklanych koralików i to taki, w którym koraliki tego samego koloru nie sąsiadują ze sobą. Nitkę już ma, ale koraliki musi kupić za swoje kieszonkowe. Ceny koralików różnych kolorów są różne, dla każdego dodatniego całkowitego n w sprzedaży są koraliki jednego koloru w cenie n gr za koralik. Tomcio zastanawia się z jakich koralików zrobić model drzewa, tak aby wydać jak najmniej.

Opis drzewa ma postać struktury listowej, w której każdy węzeł to lista jego synów. Napisz procedurę `koraliki`, która na podstawie opisu drzewa obliczy minimalny koszt koralików potrzebnych do wykonania jego modelu.

Uwaga: Dwa kolory wystarczą do wykonania modelu, ale nie dają minimalnego kosztu. Możesz założyć, że optymalna liczba kolorów nie przekracza $\log_2 n$, gdzie n to liczba koralików.

- [IX OI] Zadanie „Waga”. Dany jest zestaw odważników. Czy można na szalkach wagi położyć część odważników tak, żeby waga nadal była zrównoważona? Jeśli tak, to jaki najcięższy odważnik można przy tym użyć?
- [IX OI] Zadanie „Nawiasy”. Dane jest wyrażenie postaci $x_1 \pm x_2 \pm \dots \pm x_n$. Na ile różnych sposobów można uzyskać wyrażenie równoważne danemu wstawiając nawiasy do wyrażenia $x_1 - x_2 - \dots - x_n$? Uwaga: wstawiamy $n - 1$ par nawiasów w pełni określając kolejność wykonywania odejmowań.

19. [XI OI] Zadanie „Sznurki”. Interesują nas modele drzew (spójnych grafów acyklicznych) wykonane ze sznurka. Każde drzewo składa się z wierzchołków i pewnej liczby krawędzi łączących różne wierzchołki. Ten sam wierzchołek może być połączony z wieloma innymi wierzchołkami. Wierzchołki grafów mają być modelowane przez węzły zawiązane na kawałkach sznurka, a krawędzie przez odcinki sznurka pomiędzy węzłami. Każdy węzeł może być wynikiem zasuplenia kawałka sznurka lub związania w tym samym miejscu wielu kawałków. Każda krawędź ma długość 1. Długość sznurka użytego do zrobienia węzłów jest pomijalna.

Chcemy zoptymalizować liczbę i długość kawałków sznurka użytych do wykonania modelu drzewa:

- w pierwszym rzędzie należy zminimalizować liczbę potrzebnych kawałków sznurka,
- zakładając, że liczba kawałków sznurka jest minimalna, należy zminimalizować długość najdłuższego kawałka sznurka.

Wytyczne dla prowadzących ćwiczenia

W zadaniu 6 o wróblach, dla każdego węzła drzewa należy obliczyć: posortowaną listę możliwych ciężarów poddrzewa, przy których jest ono zrównoważone, oraz łączną masę poddrzewa bez wróbli. Łącząc dwa poddrzewa scalamy dwie posortowane listy, zostawiając tylko te elementy, które występują na obydwu listach. Dodatkowo, jeżeli ciężary obu poddrzew są równe i występują na obu listach, to można rzucić kamieniem we wspólnego ojca obu drzew i umieszczamy wagę obu poddrzew i ojca na liście wynikowej.

Fakt: Dla dowolnego wierzchołka, długość obliczonej dla niego listy nie przekroczy długości najkrótszej ścieżki od niego do jakiegoś liścia.

Analizując koszt algorytmu możemy przyjąć, że przetworzenie każdego wierzchołka zajmuje czas proporcjonalny do długości wyznaczonej dla niego listy (+1). Następnie, koszt przypadający na dany wierzchołek rozpraszamy przypisując go wierzchołkom z poddrzewa, którego jest korzeniem: na dany wierzchołek przypada koszt 1, na jego synów po $\frac{1}{2}$, na wnuki po $\frac{1}{4}$ itd. Odwróćmy teraz problem i policzmy, jaki łącznie koszt może być przypisany jednemu wierzchołkowi. Nie będzie to więcej niż: $1 + \frac{1}{2} + \frac{1}{4} + \dots \leq 2$. Stąd, łączny koszt czasowy algorytmu wynosi $\Theta(n)$.

Wykład 20. Kody Huffmana i algorytmy zachłanne

20.1 Rodzaje kodów

- Kody stałej długości (np. ASCII).
- Kody zmiennej długości (np. Morse'a).
- Problem separatora — kody prefiksowe.

Przykład: [SICP] Kod zmiennej długości może być efektywniejszy niż kod stałej długości.

A 000	C 010	E 100	G 110
B 001	D 011	F 101	H 111

Przy takim kodowaniu, wiadomość:

BACADAEAFABBAAGAH

jest zakodowana jako ciąg 54 bitów:

001000010000011000100000101000001001000000000110000111

A 0	C 1010	E 1100	G 1110
B 100	D 1011	F 1101	H 1111

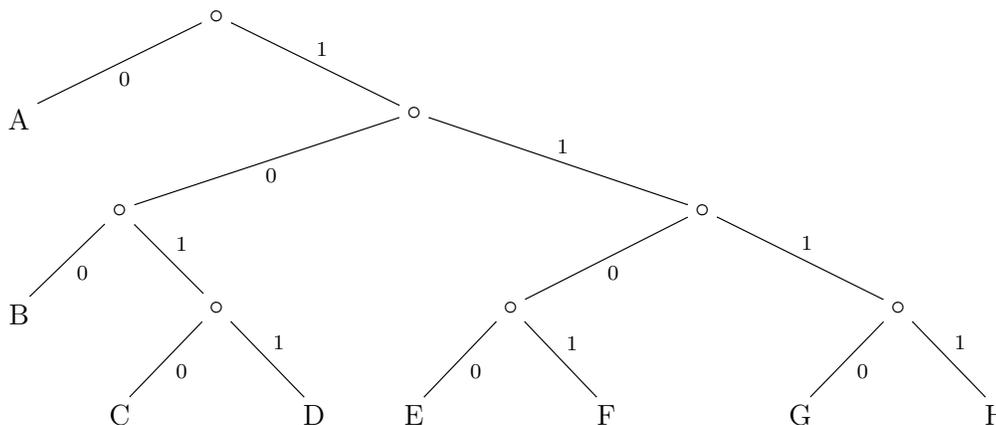
Przy takim kodowaniu, ta sama wiadomość jest zakodowana jako:

100010100101101100011010100100000111001111

Ten ciąg składa się z 42 bitów, a więc, w porównaniu z kodem stałej długości przedstawionym powyżej, oszczędza ok. 20% pamięci.

- Problem zdefiniowania optymalnego kodu prefiksowego przy znanych (względnych) częstościach występowania znaków.
- Reprezentacja optymalnego kodowania w postaci regularnego drzewa binarnego.

Przykład: Kod z poprzedniego przykładu możemy przedstawić jako drzewo. To, że kod jest prefiksowy odpowiada temu, że kodowane znaki odpowiadają liściom.



20.2 Algorytm Huffmana

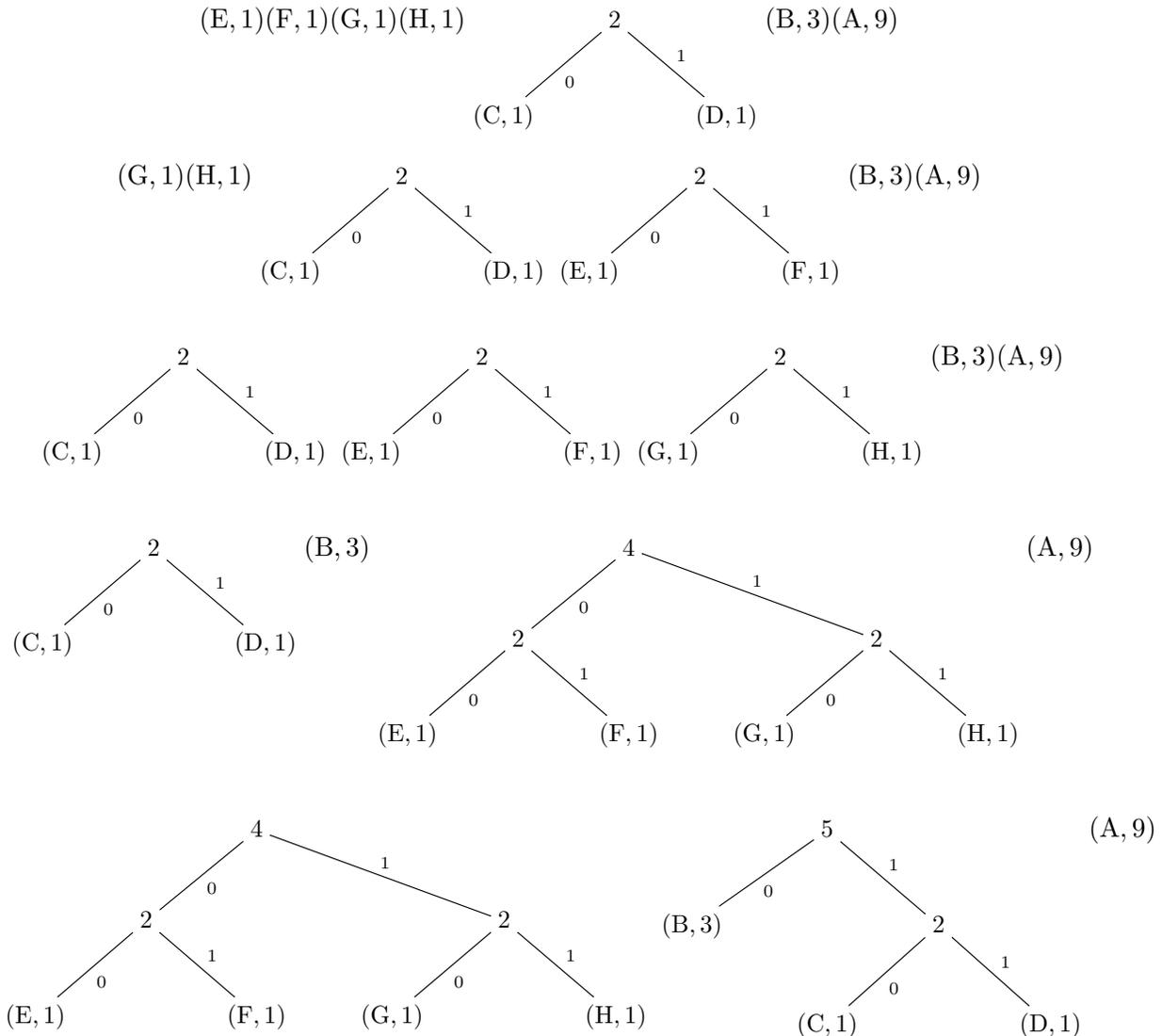
Algorytm Huffmana polega na budowaniu drzewa kodowania od liści do korzenia. Na początku znamy zbiór liści — jest to zbiór wszystkich kodowanych znaków.

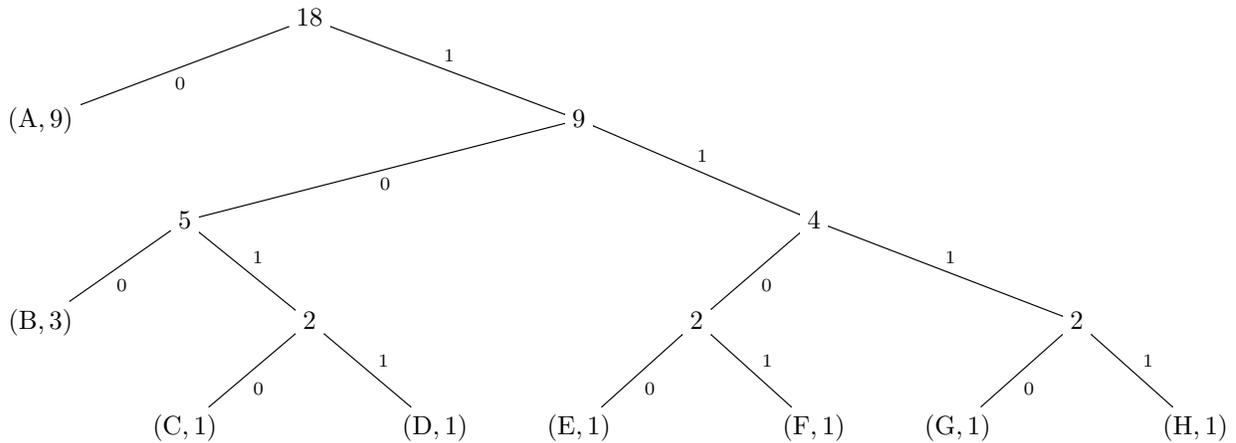
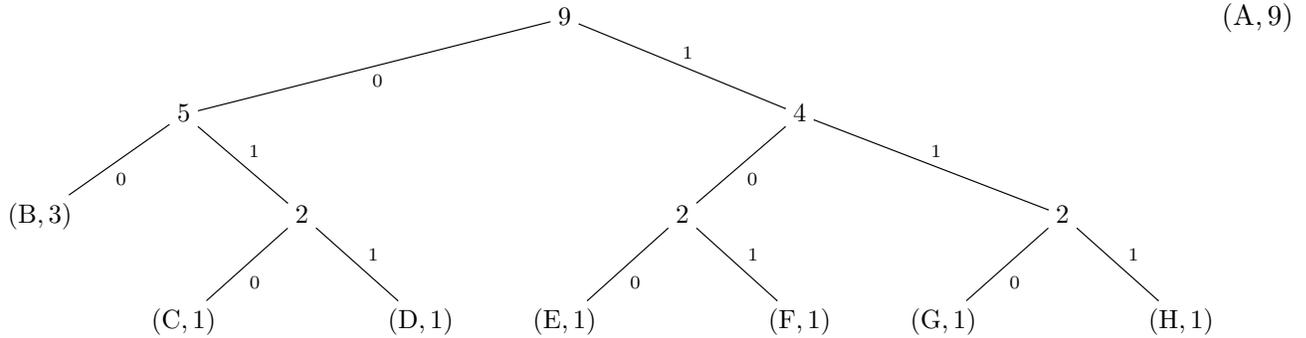
W każdym kolejnym kroku wybieramy dwa elementy o najmniejszych częstościach występowania i skleamy je, tak że ich kody będą się różnić ostatnim bitem. Tak powstały nowy element zastępuje sklezione elementy, a jego częstość występowania jest równa sumie częstości występowania sklejonych elementów.

Algorytm kończymy, gdy zostanie nam już tylko jeden element — drzewo kodowania.

Przykład: Prześledźmy działanie algorytmu dla przykładowych danych.

(C, 1)(D, 1)(E, 1)(F, 1)(G, 1)(H, 1)(B, 3)(A, 9)





Dowód poprawności:

Lemat [CLR] Niech C będzie alfabetem, a każdy znak $c \in C$ występuje $f(c)$ razy. Niech x i y będą parą znaków z C o najmniejszej liczbie wystąpień. Istnieje wtedy optymalny kod prefiksowy dla C , w którym kody dla x i y mają tę samą długość i różnią się tylko ostatnim bitem.

Dowód: Rozważmy pewne optymalne kodowanie. Niech b i c będą dwoma sąsiednimi liśćmi o maksymalnej głębokości $h_b = h_c$. Bez straty ogólności możemy założyć, że $f(b) \leq f(c)$ oraz $f(x) \leq f(y)$. Ponadto mamy $f(x) \leq f(b)$ i $f(y) \leq f(c)$, a głębokości x i y nie przekraczają głębokości b i c , $h_x, h_y \leq h_b = h_c$. Zamieniamy b z x i c z y . Długość kodu zmienia się o:

$$f(x)(h_b - h_x) + f(y)(h_c - h_y) + f(b)(h_x - h_b) + f(c)(h_y - h_c) =$$

$$(f(x) - f(b))(h_b - h_x) + (f(y) - f(c))(h_c - h_y) \leq 0$$

Tak powstałe drzewo daje nie gorsze kodowanie, a więc również optymalne.

Lemat Niech C będzie alfabetem, f będzie funkcją określającą częstość występowania symboli, x i y będą różnymi symbolami o najmniejszych częstościach występowania. Niech z będzie nowym symbolem. Przyjmijmy $f(z) = f(x) + f(y)$. Niech T będzie regularnym drzewem binarnym odpowiadającym optymalnemu kodowaniu dla alfabetu $(C \setminus \{x, y\}) \cup \{z\}$ i f .

Wówczas drzewo T' powstałe przez dodanie do wierzchołka z synów x i y jest optymalnym drzewem kodowania dla C i f .

Dowód: Gdyby T' nie było optymalne, to istniałoby lepsze drzewo kodowania, w którym x i y byłiby braćmi. Oznaczając ich ojca przez z i usuwając je, uzyskalibyśmy drzewo kodowania lepsze od T , co nie jest możliwe.

Z tych dwóch lematów wynika natychmiast poprawność algorytmu.

20.3 Implementacja algorytmu Huffmana

Zgodnie z poprzednim opisem implementujemy algorytm Huffmana:

```
let huffman l =
  let rec process col =
    if huff_size col = 1 then
      huff_first col
    else
      let t1 = huff_first col
      and col1 = huff_remove col
      in
        let t2 = huff_first col1
        and col2 = huff_remove col1
        in
          process (huff_put col2 (merge t1 t2))
  in
    process (make_huff_col l);;
```

Zakładamy tu, że elementy listy l są posortowane niemalejąco wg częstości występowania. Korzystamy tu z drzew Huffmana z operacją `merge` oraz kolejek priorytetowych drzew Huffmana, z operacjami: `make_huff`, `huff_size`, `huff_first`, `huff_put` i `huff_remove`.

20.4 Implementacja drzew Huffmana

```
type 'a huff_tree =
  Letter of ('a * float) |
  Node of ('a huff_tree * 'a huff_tree * float);;

let frequency (t : 'a huff_tree) =
  match t with
  Letter (_,f) ->f |
  Node (_,_,f)   ->f;;

let merge t1 t2 = Node (t1,t2,frequency t1 +. frequency t2);;
```

20.5 Implementacja kolekcji drzew Huffmana

```
type 'a huff_col = 'a huff_tree fifo * 'a huff_tree fifo;;

let make_huff_col l =
```

```

(fold_left (fun q x ->put q (Letter x)) empty_queue 1,empty_queue);;

let huff_size ((q1,q2): 'a huff_col) = size q1 + size q2;;

let huff_put ((q1,q2): 'a huff_col) t =
  ((q1,put q2 t): 'a huff_col);;

let huff_first ((q1,q2): 'a huff_col) =
  if q1 = empty_queue then first q2 else
  if q2 = empty_queue then first q1 else
  let f1 = first q1
  and f2 = first q2
  in
    if frequency f1 <= frequency f2 then f1 else f2;;

let huff_remove ((q1,q2): 'a huff_col) =
  if q1 = empty_queue then (q1,remove q2) else
  if q2 = empty_queue then (remove q1,q2) else
  let f1 = first q1
  and f2 = first q2
  in
    if frequency f1 <= frequency f2 then
      (remove q1,q2)
    else
      ((q1,remove q2): 'a huff_col);;

```

Zasymuluj działanie algorytmu Huffmana pokazując zawartość kolejek implementujących kolekcję drzew.

20.6 Schemat programowania zachłannego

- szukamy pewnego rozwiązania optymalnego, robimy to za pomocą ciągu wyborów („lokalnych”) kolejnych elementów rozwiązania; analogia ze stertą cegieł i zbudowaniem jak największego domu z 1000 cegieł,
- własność wyboru zachłannego — ciąg wyborów optymalnych lokalnie prowadzi nas do rozwiązania optymalnego globalnie,
- jeśli konstruujemy optymalne rozwiązanie przez dokonywanie kolejnych wyborów, to taką własność zwykle dowodzimy indukcyjnie: pokazujemy, że jeśli istnieje rozwiązanie optymalne zgodne z dokonanymi do tej pory wyborami, to istnieje rozwiązanie optymalne zgodne również z kolejnym wyborem (wliczając w to sytuację, gdy dokonujemy pierwszego wyboru),
- jeśli konstruujemy programy zgodnie z zasadą „dziel i rządź”, to zwykle własności zachłannego wyboru dowodzimy w oparciu o własność optymalnej podstruktury — optymalne rozwiązanie jest funkcją optymalnych rozwiązań podproblemów i łączącego je zachłannego wyboru.

Ćwiczenia

1. [CLR] Ciągły problem plecakowy.
2. [CLR, p. 17.1] Problem wyboru zajęć (historyjka o kinomaniu i jak największej liczbie filmów do obejrzenia).
3. Dany jest zbiór odcinków na prostej. Jaka jest minimalna liczba gwoździ, za pomocą których można je przybić do prostej? Każdy odcinek musi być przybity przynajmniej jednym gwoździem.
4. Rozplanować zbiór zajęć po salach. Jaka jest minimalna liczba potrzebnych sal?
5. [CLR, ćw 17.2-4] Problem tankowania na stacjach benzynowych.
6. Napisz procedurę `silnie : int → int list`, która dla danej dodatniej liczby całkowitej n znajduje najkrótszy taki ciąg dodatnich liczb całkowitych $[x_1; x_2; \dots; x_k]$, że $x_1! + x_2! + \dots + x_k! = n$.

Na przykład, dla `silnie 42` poprawnym wynikiem jest n.p. `[3; 4; 3; 3]`.

7. [PCh] Dany jest ciąg nawiasów, otwierających i zamykających. Napisz procedurę `nawiasy`, która obliczy minimalną liczbę nawiasów, które należy obrócić tak, aby uzyskać poprawne wyrażenie nawiasowe. Jeżeli nie jest to możliwe, to poprawnym wynikiem jest `-1`.

```
type nawias = Otwierający | Zamykający
```

```
val nawiasy : nawias list → int
```

8. W trakcie wakacji n informatyków pojechało razem na obóz informatyczny. Zamieszkali oni w domkach kempingowych stojących wzdłuż prostej drogi. Firma telekomunikacyjna chce im zapewnić dostęp do Internetu, rozstawiając wzdłuż drogi nadajniki sieci bezprzewodowej. Jeden nadajnik może obsługiwać co najwyżej k domków znajdujących się od niego w odległości co najwyżej r . Napisz procedurę `wakacje : int → int → int list → int`, która na podstawie liczb k i r , oraz listy współrzędnych domków obliczy minimalną liczbę potrzebnych nadajników.
9. Ciąg $[x_1; x_2; \dots]$ nazwiemy *naprzemiennym*, jeżeli:
 - $x_1 < x_2 > x_3 < x_4 > \dots$, lub
 - $x_1 > x_2 < x_3 > x_4 < \dots$

Ciąg jednoelementowy i pusty są również naprzemiennie.

Napisz procedurę `naprzemienny : int array → int`, która mając dany ciąg liczb całkowitych w postaci tablicy obliczy długość najdłuższego jego naprzemiennego podciągu.

Na przykład, `naprzemienny [|4;8;6;5;3;4;5;3;7;9;5;7|]` = 8.

10. Rozważamy ciągi liczb całkowitych (x_1, x_2, \dots, x_k) konstruowane zgodnie z następującymi regułami:

- $x_1 = 1$,
- $x_{i+1} = x_i + f_i$ dla pewnej liczby Fibonacciego f_i .

Napisz procedurę `ciag : int → int list`, która dla danej dodatniej liczby całkowitej n wyznaczy najkrótszy taki ciąg (x_1, x_2, \dots, x_k) , że $x_k = n$.

- [XII OI, zadanie Lot na marsa] Bajtazar postanowił polecieć na Marsa, aby zwiedzić istniejące tam stacje badawcze. Wszystkie stacje na Marsie leżą na okręgu. Bajtazar ląduje w jednej z nich, a następnie porusza się za pomocą specjalnego pojazdu, który jest napędzany odpowiednim paliwem. Litr paliwa starcza na metr jazdy. Zapasy paliwa są jednak niewielkie, różne jego ilości znajdują się w różnych stacjach. Bajtazar może tankować paliwo na stacji, na której w danym momencie się znajduje, nie więcej jednak niż dostępna tam jego ilość (pojemność baku jest nieograniczona). Musi mu to wystarczyć na dojazd do następnej stacji. Bajtazar musi zdecydować, gdzie powinien wylądować, tak żeby mógł zwiedzić wszystkie stacje. Na koniec Bajtazar musi wrócić do stacji, w której wylądował. W czasie podróży Bajtazar musi poruszać się po okręgu, stale w wybranym jednym z dwóch kierunków.
- Student rozwiązuje zadania ze skryptu z WPF. Każde zadanie ma dwa współczynniki (t, p) : t — trudności zadania i p — jego pracochłonności. W miarę rozwiązywania zadań, student jest coraz bardziej zmęczony. Stopień zmęczenia studenta określamy liczbą z . Student potrzebuje $tz + p$ czasu na rozwiązanie zadania, po czym jego zmęczenie rośnie o p . Zakładamy, że na początku zmęczenie studenta wynosi 0. Napisz procedurę `wpf : (int * int) list → int`, która dla listy zadań w skrypcie $[(t_1, p_1); \dots; (t_n, p_n)]$ zwróci minimalny czas na ich zrobienie. Zadania można wykonywać w dowolnej kolejności.
- Firma X podjęła się szeregu zobowiązań, z których wykonaniem zalega. Wykonanie każdego z nich zajmuje określoną liczbę dni. Za każdy dzień opóźnienia wykonania każdego z zobowiązań firma płaci określone kary. Napisz procedurę `harmonogram`, która obliczy optymalną kolejność wykonywania zaległych zobowiązań. Procedura ta otrzymuje listę trójek postaci:

(nazwa czynności, liczba dni, kara za dzień zwłoki)

Jej wynikiem powinna być lista nazw czynności w kolejności, w jakiej należy je wykonywać.

Wytyczne dla prowadzących ćwiczenia

Ad. 2 i 3 Oba problemy mają takie samo rozwiązanie. Wynika to z dualnego twierdzenia Dilworth'a (zobacz http://wazniak.mimuw.edu.pl/index.php?title=Matematyka_dyskretna_2/Wyk%C5%82ad_2:). Wprowadźmy częściowy porządek na odcinkach: jeden odcinek jest mniejszy od drugiego, jeżeli jego koniec jest mniejszy od początku drugiego. Kino-man szuka najdłuższego łańcucha, a każdy gwóźdź mocuje pewien antyłańcuch. Dualne twierdzenie Dilworth'a mówi zaś, że rozmiar największego łańcucha jest taki sam, jak minimalna liczba antyłańcuchów pokrywających cały zbiór.

Ad. 12 Rozwiązanie polega na posortowaniu zadań po $\frac{p}{t}$.

Ad. 13 Rozwiązanie polega na posortowaniu zleceń po $\frac{\text{czas wykonania}}{\text{kara}}$.

Wykład 21. Algorytmy zachłanne c.d.

21.1 Problem kajakowy

Problem⁷: Jak usadzić n osób o podanych wagach w minimalnej liczbie kajaków? Kajaki są dwumiejscowe i mają określoną wyporność. W kajaku może płynąć tylko jedna lub dwie osoby. Zakładamy, że waga żadnej z osób nie przekracza wyporności kajaka, ale dwie osoby mogą już przekroczyć wyporność kajaka. Wagi osób są dane w postaci listy uporządkowanej niemalejąco.

Problem ten możemy rozwiązać zachłannie i to na kilka sposobów. W rozwiązaniach tych będziemy korzystać z dwustronnej kolejki FIFO-LIFO. Dla uproszczenia zakładamy, że dane są posortowane niemalejąco.

1. Najgrubszego kajakarza nazwiemy grubasem. Tych spośród pozostałych kajakarzy, którzy mieszczą się z nim w kajaku, nazwiemy chudzielcami. Pozostałych również nazwiemy grubasami.

Idea algorytmu polega na tym, że największemu grubasowi znajdujemy jak najgrubszego chudzielca, który może płynąć razem z nim. Czyli sadzamy razem największego grubasa i największego chudzielca. Zauważmy przy tym, że im chudszy grubas, tym grubszy może być chudzielec. Tak więc podział na grubasów i chudzielców będzie zmieniał się w czasie — chudsze grubasy będą przechodzić do puli chudzielców, a w razie braku grubasów, najgrubszy chudzielec może zostać uznany za grubasa.

Początkowo o wszystkich zakładamy, że są grubi i korygujemy podział.

```
let kajaki l wyp =
  let rec dobierz g ch =
    if (is_empty_queue g) && (is_empty_queue ch) then
      (g,ch) else
    if is_empty_queue g then
      (make_queue [last ch],remove_last ch) else
    if queue_size g = 1 then (g,ch) else
    if first g + last g <= wyp then
      dobierz (remove_first g) (put_last ch (first g))
    else (g,ch)
  in
  let rec sadzaj gp chp acc =
    let (g,ch) = dobierz gp chp
    in
      if is_empty_queue g then acc else
      if is_empty_queue ch then
```

⁷Podane sformułowanie problemu pochodzi z zadania *Kajaki* z IV OI. Problem ten został wcześniej sformułowany niezależnie przez autora w równoważnej postaci.

```

        sadzaj (remove_last g) ch ([last g]::acc)
    else
        sadzaj
            (remove_last g)
            (remove_last ch)
            ([last g; last ch]::acc)
    in
        sadzaj (make_queue 1) empty_queue [];;

```

```

kajaki [1; 2; 2; 3; 3; 3; 4; 6; 6; 8; 8; 8; 9; 9; 10] 10;;
[[3; 3]; [6; 3]; [6; 4]; [8]; [8; 2]; [8; 2]; [9]; [9; 1]; [10]]

```

Uzasadnienie poprawności:

- Istnieje optymalne rozwiązanie, w którym największy grubas g i najgrubszy chudzielec c płyną razem.
 - Jeśli g płynie sam, to można do niego dosadzić c .
 - Załóżmy, że g płynie z jakimś x , więc $x \leq c$. c i x można zamienić miejscami.
 - Własność optymalnej podstruktury — oczywista. Po obsadzeniu pierwszego kajaka, pozostali kajakarze powinni płynąć minimalną liczbą kajaków.
2. Najchudszy kajakarz jest chudzielcem. Grubasy, to ci, którzy nie mieszczą się z nim w kajaku. Pozostali to chudzielce. Jeśli jest tylko jeden chudy, to przyjmujemy, że jest on gruby.

Idea algorytmu polega, żeby najchudszego chudzielca posadzić z jak najgrubszym kajakarzem, czyli najgrubszym chudzielcem. Początkowo o wszystkich zakładamy, że są chudzi, a następnie przrzucamy ich do puli grubych. W miarę usadzania w kajakach część chudych przechodzi do puli grubych. Kajakarze, którzy raz trafią do puli grubych na pewno będą siedzieć samotnie w kajaku. Na koniec grubasy siadają samotnie.

```

let kajaki l wyp =
    let rec dobierz ch g =
        if is_empty_queue ch then (ch,g) else
        if queue_size ch = 1 then
            (empty_queue,put_first g (last ch)) else
        if first ch + last ch >wyp then
            dobierz (remove_last ch) (put_first g (last ch))
        else
            (ch,g)
    in
        let rec sadzaj chp gp acc =

```

```

let (ch,g) = dobierz chp gp
in
  if (is_empty_queue ch) && (is_empty_queue g) then acc else
  if is_empty_queue ch then
    sadzaj ch (remove_first g) ([first g]::acc)
  else
    sadzaj (remove_first (remove_last ch)) g
      ([first ch; last ch]::acc)
in
  sadzaj (make_queue 1) empty_queue [];;

kajaki [1; 2; 2; 3; 3; 3; 4; 6; 6; 8; 8; 8; 9; 9; 10] 10;;
[[10]; [9]; [8]; [3; 4]; [3; 6]; [3; 6]; [2; 8]; [2; 8]; [1; 9]]

```

Uzasadnienie poprawności:

- Jeśli brak chudzielców, to każdy płyne osobno.
 - Jeśli są chudzielcy, to jest ich przynajmniej dwóch. Wówczas istnieje optymalne rozwiązanie, w którym najgrubszy g i najchudszy chudzielec c płyną razem.
 - Jeśli c płyne sam, to można do niego dosadzić g .
 - Załóżmy, że c płyne z jakimś x , więc $x \leq g$. g i x można zamienić miejscami.
 - Własność optymalnej podstruktury — oczywista. Po obsadzeniu pierwszego kajaka, pozostali kajakarze powinni płynąć minimalną liczbą kajaków.
3. Najgrubszego kajakarza sadzamy z najchudszym, o ile się zmieszczą. Jeśli nie, to najgrubszy kajakarz płyne sam.

```

let kajaki l wyp =
  let rec sadzaj q acc =
    if is_empty_queue q then acc else
    if queue_size q = 1 then [first q]::acc else
    if first q + last q <= wyp then
      sadzaj (remove_first (remove_last q)) ([first q; last q]::acc)
    else
      sadzaj (remove_last q) ([last q]::acc)
  in
    sadzaj (make_queue 1) [];;

kajaki [1; 2; 2; 3; 3; 3; 4; 6; 6; 8; 8; 8; 9; 9; 10] 10;;
[[3; 4]; [3; 6]; [3; 6]; [8]; [2; 8]; [2; 8]; [9]; [1; 9]; [10]]

```

Uzasadnienie poprawności.

To rozwiązanie generuje takie same wyniki jak poprzednie (z dokładnością do kolejności elementów na liście). Dlaczego?

4. Możemy też wsadzać do kajaka parę dwóch kolejnych, jak najgrubszych. Grubi są ci, którzy nie zmieszczą się do jednego kajaka razem z kolejnym chudszy. Najchudszy jest z definicji chudy. Sadzamy razem dwóch najgrubszych chudych. Stopniowo grubi mogą przenosić się do puli chudych.

Tym razem zamiast kolejek wystarczą nam listy. Lista grubych jest uporządkowana niemalejąco, a chudych nierosnąco.

```
let kajaki l wyp =
  let rec dobierz ch g =
    match (ch,g) with
    | (_, []) ->(ch, []) |
    | ([],h::t) ->dobierz [h] t |
    | (chh::cht,gh::gt) ->
      if chh + gh <= wyp then
        dobierz (gh::ch) gt
      else
        (ch,g)
  in
  let rec sadzaj chp gp acc =
    let (ch,g) = dobierz chp gp
    in
    match ch with
    | [] ->acc |
    | [h] ->sadzaj [] g ([h]::acc) |
    | h1::h2::t ->sadzaj t g ([h2; h1]::acc)
  in
  sadzaj [] l [];;

kajaki [1; 2; 2; 3; 3; 3; 4; 6; 6; 8; 8; 8; 9; 9; 10] 10;;
[[10]; [9]; [9]; [1; 8]; [2; 8]; [2; 8]; [3; 3]; [3; 6]; [4; 6]]
```

Uzasadnienie poprawności.

Ćwiczenia

Zadania rozwiązuje się (mniej lub bardziej) zachłannie. Dla każdego z nich, oprócz programu, należy uzasadnić poprawność rozwiązania zachłannego.

- [II OI, zadanie Palindromy] Podziel dane słowo (listę znaków) na maksymalną liczbę palindromów parzystej długości.
- [V OI, zadanie Suma ciągu jedynkowego (uproszczone)] Dana jest liczba $n \geq 0$ oraz $0 \leq s \leq \frac{n(n+1)}{2}$. Podaj podzbiór zbioru $\{1, \dots, n\}$, którego suma elementów jest równa s . Czy jest to zbiór o minimalnej liczbie elementów?
- Napisz procedurę `podzial : int → int list`, która dla danej nieujemnej liczby całkowitej przedstawi ją jako sumę jak największej liczby (dodatnich) różnych liczb Fibonacciego. Na przykład, `podzial 42 = [1; 2; 5; 13; 21]`.
- [IV OI, zadanie Lotniska] W n miastach (ponumerowanych od 1 do n) znajdują się lotniska. Każde lotnisko ma określoną przepustowość, tj. liczbę połączeń z innymi miastami. Połączenia takie są zwrotne, czyli połączenie z A do B jest jednocześnie połączeniem z B do A . Podaj połączenia realizujące **dokładnie** podane przepustowości lub stwierdź, że się nie da. Dane mają postać listy uporządkowanej niemalejąco wg przepustowości miast.
 - wersja łatwiejsza: między dwoma miastami może być wiele połączeń,
 - wersja trudniejsza: między dwoma miastami może być co najwyżej jedno połączenie.
- [Zadanie Wektorki, wersja 1D; podobne do zadania Steps (nr 846) z UVA] Tworzymy ciągi liczb całkowitych (x_k) i (v_k) w następujący sposób:
 - $x_0 = 1, v_0 = 0,$
 - dla każdego $k > 0$ wybieramy pewne v_k spełniające $v_{k-1} - 1 \leq v_k \leq v_{k-1} + 1$ oraz przyjmujemy $x_k = x_{k-1} + v_k$.

Napisz procedurę `wyścig : int → int list`, która dla danego $n \geq 1$ znajdzie **najkrótszy** taki ciąg x_0, x_1, \dots, x_k , który spełnia powyższe warunki, oraz $x_k = n$. Wynikiem procedury powinna być lista $[x_0; \dots; x_k]$.

- Dana jest lista par liczb postaci (x, y) , $x < y$, opisujących (domknięte) odcinki alejki jakie mogą oświetlić poszczególne latarnie, oraz para liczb (a, b) , $a < b$, opisująca odcinek alejki, który chcemy oświetlić.

Napisz procedurę `ile : (int * int) list → (int * int) → int`, która obliczy minimalną liczbę latarni, jakie są konieczne do oświetlenia danego odcinka alejki.

Przykład:

```
ile [(1, 4); (7, 9); (-2, 3); (4, 9); (9, 12); (0, 5); (3, 7)] (-1, 11) = 4
```

- [X OI, zadanie Czekolada] Dana jest tabliczka czekolady, którą należy połamać na cząstki. Każde przełamanie kawałka czekolady jest obciążone pewnym kosztem. Koszt

ten nie zależy od wielkości kawałka czekolady, ale od prostej, wzdłuż której łamiemy. Dla wszystkich prostych, na których leżą krawędzie cząstek czekolady mamy stałe określające koszty łamania.

8. [III OI, zadanie Wieże] Na szachownicy $n \times n$ należy ustawić n wież tak, żeby żadne dwie się nie atakowały, a ponadto każda z tych wież ma wyznaczony prostokąt, w którym ma stać. Znajdź takie ustawienie lub stwierdź, że się nie da.
9. [II OI, zadanie Obchodzenie drzewa skokami] Jak obejść wszystkie wierzchołki drzewa (ogólnego) tak, aby odległość między kolejnymi wierzchołkami nie przekraczała 3.
10. [XI OI, zadanie Most] Dana jest grupa osób, które chcą przejść nocą przez most. Mają oni jedną latarkę, która pozwala przejść jednej lub dwóm osobom. Każda osoba potrzebuje określonego czasu na przejście przez most. (Możesz założyć, że dane są posortowane.) Jeśli dwie osoby idą razem, to potrzebują tyle czasu, co wolniejsza z nich. Jaki jest minimalny czas potrzebny do przejścia wszystkich?
11. [XII OI, zadanie Samochodziki; OSC, optymalna strategia wymiany stron] Jasio jest trzylatkiem i bardzo lubi bawić się samochodzikami. Jasio ma n różnych samochodzików. Wszystkie samochodziki leżą na wysokiej półce, do której Jasio nie dosięga. W pokoju jest mało miejsca, więc na podłodze może znajdować się jednocześnie co najwyżej k samochodzików.

Jasio bawi się jednym z samochodzików leżących na podłodze. Oprócz Jasia w pokoju cały czas przebywa mama. Gdy Jasio chce się bawić jakimś innym samochodzikiem i jeśli ten samochodzik jest na podłodze, to sięga po niego. Jeśli jednak samochodzik znajduje się na półce, to musi mu go podać mama. Mama, podając Jasiowi jeden samochodzik, może przy okazji zabrać z podłogi dowolnie wybrany przez siebie inny samochodzik i odłożyć go na półkę (tak, aby na podłodze nie brakło miejsca).

Mama bardzo dobrze zna swoje dziecko i wie dokładnie, którymi samochodzikami Jasio będzie chciał się bawić. Dysponując tą wiedzą, mama chce zminimalizować liczbę przypadków, gdy musi podawać Jasiowi samochodzik z półki. W tym celu musi bardzo rozważnie odkładać samochodziki na półkę.

Napisz procedurę `samochodziki`, która dla danej liczby k oraz listy samochodzików, którymi zechce się bawić Jasio, obliczy minimalną liczbę razy, kiedy mama będzie musiała Jasiowi podać samochodzik.

12. [XV OI, zadanie Plakatowanie] Rozważamy następującą łamigłówkę. Dana jest tablica nieujemnych liczb całkowitych $a = [a_1; \dots; a_n]$. Jeśli $a_i = a_{i+1} = \dots = a_{j-1} = a_j$ (dla $1 \leq i \leq j \leq n$), to w jednym ruchu można zmniejszyć wszystkie liczby a_i, a_{i+1}, \dots, a_j o dowolną dodatnią liczbę, nie większą niż a_i (tak żeby po zmniejszeniu nadal były nieujemne). Napisz procedurę:

`ruchy : int array → (int * int * int) list`

która dla danej tablicy a wyznaczy najkrótszą sekwencję ruchów potrzebnych do wyzerowania wszystkich liczb a_1, \dots, a_n . Trójka liczb (i, j, k) reprezentuje zmniejszenie liczb a_i, \dots, a_j o k .

Wytyczne dla prowadzących ćwiczenia

- Ad. 1** Oczekujemy złożoności czasowej $O(n^2)$.
- Ad. 3** To zadanie można zrobić zachłannie na kilka sposobów. Najprościej jest wyznaczyć najmniejsze takie s , że $s = Fib_2 + Fib_3 + \dots + Fib_k \geq n$, gdzie n jest zadaną liczbą. Wówczas spośród liczb $Fib_2, Fib_3, \dots, Fib_k$ należy usunąć jak najmniej liczb Fibonacciego, które w sumie dają $s - n$. A to można już robić zachłannie, usuwając w każdym kroku największą liczbę, która nie jest zbyt duża.
- Ad. 7** Można udowodnić, że istnieje rozwiązanie optymalne, w którym łamiemy kawałki przez całą szerokość tabliczki czekolady.
- Ad. 8** Wskazówka: problem sprowadza się do dwóch problemów jednowymiarowych.
- Ad. 11** W tym zadaniu, za historyjką kryje się problem optymalnej strategii wymiany stron w pamięci wirtualnej.

Wykład 22. Algorytm mini-max

Problem: jak zaprogramować optymalną strategię grania w grę dla dwóch graczy? Możemy tu zastosować technikę podobną do back-tracking, przeszukując wszystkie możliwe sekwencje ruchów. Sekwencje te utworzą drzewo. Jeśli gra musi się kiedyś zakończyć, to drzewo to ma skończoną wysokość. Jeśli przy tym za każdym razem gracz ma do wyboru skończoną liczbę ruchów, to całe drzewo jest skończone, a więc można je przejrzeć.

Jakie kryteria powinniśmy przyjąć, wybierając ruchy? Każda sytuacja końcowa oznacza wygraną jednego z graczy lub remis. Wygrana ponadto może być większa lub mniejsza. Może to odpowiadać liczbie wygranych punktów i/lub pieniędzy. Sytuacje te możemy ocenić za pomocą funkcji przypisującej im wagi rzeczywiste. Wagi dodatnie oznaczają wygraną jednego gracza, wagi ujemne drugiego, a zero to remis.

W ten sposób przypisaliliśmy liściom wagi oceniające sytuacje końcowe. Możemy też przypisać podobne wagi oceniające sytuację w trakcie gry, przy założeniu, że obaj gracze grają optymalnie. Co to jednak znaczy, że gracze grają optymalnie? Co to jest optymalna strategia?

Mówimy, że w danej sytuacji gracz ma strategię prowadzącą do określonego rezultatu, jeżeli bez względu na to, jak będzie grał drugi gracz, może on osiągnąć wynik nie gorszy od danego, oraz drugi gracz może dobrać swoje ruchy tak, aby osiągnąć taki wynik. Optymalna strategia to taka, która prowadzi do najlepszego wyniku.

Twierdzenie Węzłom drzewa gry możemy przypisać wagi oznaczające wyniki optymalnych strategii zgodnie z następującą rekurencyjną regułą:

- waga liścia, to waga oceniająca sytuację końcową,
- jeśli węzeł drzewa odpowiada ruchowi wykonywanemu przez gracza chcącego uzyskać jak największy wynik, to waga ta jest równa maksimum z wag jego synów,
- jeśli węzeł drzewa odpowiada ruchowi wykonywanemu przez gracza chcącego uzyskać jak najmniejszy wynik, to waga ta jest równa minimum z wag jego synów.

Dowód Indukcyjny ze względu na wysokość drzewa.

Jak wykorzystać tak uzyskane wagi? Waga przypisana korzeniowi rozstrzyga, który z graczy ma strategię wygrywającą.

Jak wykorzystać ten algorytm do zaimplementowania optymalnej strategii gry? Należy obliczyć wagi dla drzewa gry zaczynającego się w aktualnym stanie gry i zobaczyć jaki pierwszy ruch realizuje wagę tego węzła.

22.1 Algorytm mini-max

Sam algorytm postaramy się zapisać w sposób niezależny od rodzaju gry. Graczy będziemy reprezentować przez liczby $+1$ i -1 , w zależności od tego, czy grają na max, czy na min.

```
let i = 1;;
```

```
let other who = -who;;
```

```
let rec minimax s who =
```

```

if finished s who then
  (result s who,pass)
else
  let make_move m =
    let (w,_) = minimax (move s who m) (other who)
    in (who * w,m)
  in
    let (w,m) = choose (map make_move (moves s who))
    in (w * who,m);;

```

Korzystamy tu z pomocniczej procedury wybierającej optymalny ruch:

```

let choose l =
  match l with
  [] ->failwith "Pusta lista ruchów!" |
  (h::t) ->
    fold_left
      (fun (w,m) (w1,m1) ->if w1 >w then (w1,m1) else (w,m))
      h t;;

```

```

let game s =
  let (_,m) = minimax s 1
  in m;;

```

22.2 Przykład — gra w kamienie

Zilustrujmy na przykładzie prostej gry w kamienie brakujące elementy. Każdy z graczy usuwa z kupki 1–3 kamieni. Wygrywa ten, który wykona ostatni ruch.

```

let moves s who =
  match s with
  0 ->[] |
  1 ->[1] |
  2 ->[1; 2] |
  _ ->[1; 2; 3];;

let finished s who = moves s who = [];;

let pass = 0;;

let result s who = other who;;

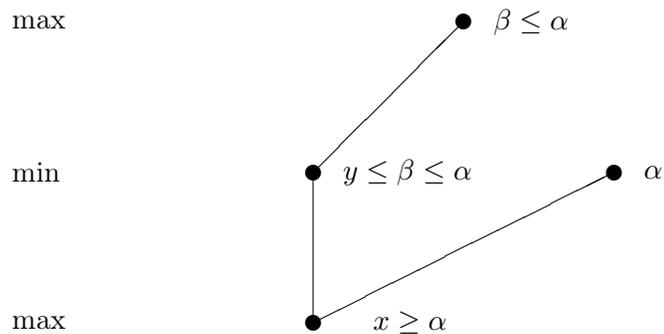
let move s who m = s - m;;

```

22.3 α - β obcięcie

[[Uwzględnić wpływ α o 4 poziomy i głębiej.]] Algorytm mini-max można polepszyć. Za-uważmy, że jeżeli w drzewie gry jakiś węzeł ma wagę α , a jego ojciec stara się zmaksymalizować (zminimalizować) wynik gry, to waga ojca x spełnia $x \geq \alpha$ ($x \leq \alpha$).

Rozważmy dalej sytuację pokazaną na rysunku:



Zauważmy, że waga węzła x nie może być mniejsza niż α . Podobnie, waga węzła y nie może być większa niż β . Jeżeli ponadto $\beta \leq \alpha$, to $y \leq \alpha$, a tym samym całe poddrzewo gry zaczepione w y nie ma najmniejszego znaczenia dla wagi x !

Algorytm α - β obcięcia polega na pomijaniu poddrzew drzewa gry, które zgodnie z powyższą regułą nie mają wpływu na wagi przypisane węzłom znajdującym się powyżej:

- wywołując rekurencyjnie funkcję obliczającą wagi, przekazujemy najlepszą do tej pory wyznaczoną wagę ojca (α),
- w momencie, gdy najlepsza aktualna waga syna (β) będzie lepsza lub równa wadze ojca, to możemy przerwać obliczenie wagi syna, gdyż nie będzie ona miała wpływu na wagę ojca.

```
exception Obciecie of int * move;;
```

```
let rec alfabet a s who alpha =
  if finished s who then (result s who,pass) else
  let choose mvs =
    try
      fold_left
        (fun (beta,mb) m ->
          let (w,_) = alfabet (move s who m) (other who) beta
          in
          if w * who >= alpha * who then raise (Obciecie (w,m)) else
          if w * who >beta * who then (w,m) else (beta,mb))
        (no_result who,pass)
      mvs
    with Obciecie (beta,mb) ->(beta,mb)
  in
  choose (moves s who);;
```

```

let game s =
  let (_,m) = alfabeto s i (no_result (other i))
  in m;;

```

Korzystamy tu z pewnych dodatkowej procedury zależnej od gry:

```

let no_result who = -2 * who;;

```

Dalsze usprawnienie można uzyskać poprzez dodanie spamiętywania. Często ta sama sytuacja w grze może być osiągnięta na wiele sposobów. Oznacza to, że to samo poddrzewo może pojawiać się wielokrotnie w grze. Zamiast wielokrotnie przeprowadzać dla niego te same obliczenia, stosujemy technikę spamiętywania:

```

let tab = ref empty;;

```

```

let rec alfabeto s who alpha =
  memoize tab (function (s,who,alpha) ->
    ...

```

W przypadku gry w kamienie, spamiętywanie ogranicza koszt do liniowego ze względu na liczbę kamieni.

22.4 Analiza algorytmu α - β obcięcia

Efektywność tego algorytmu istotnie zależy od kolejności rozpatrywanych ruchów. Ruchy należy, w miarę możliwości, starać się rozpatrywać od lepszych do gorszych. W ten sposób większa część gałęzi jest obcinana. Z braku innych kryteriów możemy też wybierać najpierw ruchy prowadzące szybciej do zakończenia rozgrywki. W ten sposób mamy większe szanse na obcięcie gałęzi, których przejrzenie jest bardzo pracochłonne.

Zastanówmy się jednak, jakiego rzędu oszczędności może przynieść α - β obcięcie w porównaniu z algorytmem mini-max. Zauważmy, że na każdym poziomie drzewa gry, pierwsze wywołanie rekurencyjne dokona obliczenia wszystkich podgałęzi. Natomiast w kolejnych wywołaniach rekurencyjnych mamy coraz większe szanse na obcinanie gałęzi.

Założmy więc, że pierwsze wywołanie rekurencyjne (gracza grającego na max.) dało wynik A . Obcięcie ma taką naturę, że jeżeli napotkamy gałąź pozwalającą na obcięcie, to wszystkie dalsze gałęzie są obcinane. Ile więc gałęzi (średnio) będzie wywoływanych? Nie obliczymy tu dokładnego wyniku, zwłaszcza, że zależy on od konkretnej gry, przyjmiemy jednak pewne upraszczające założenia. Założmy, że na każdym poziomie mamy n gałęzi do rozpatrzenia. Oznaczmy gałęzie wychodzące z kolejnego wywołania rekurencyjnego (gracza grającego na min.) przez B_1, \dots, B_n . Założmy, że prawdopodobieństwo, iż $B_i > A$ jest równe p (niezależnie od A oraz i). Wówczas prawdopodobieństwo, że będziemy obliczać wagę i gałęzi wynosi:

- $p^{i-1}(1-p)$, dla $i < n$, oraz
- p^{n-1} , dla $i = n$.

Tak więc oczekiwana liczba obliczanych wag gałęzi wynosi:

$$np^{n-1} + \sum_{k=1}^{n-1} \left(kp^{k-1}(1-p) \right) = np^n + \sum_{k=1}^n \left(kp^{k-1}(1-p) \right) \leq (1-p) \sum_{k=0}^{\infty} \left((k+1)p^k \right) =$$

$$(1-p) \sum_{k=0}^{\infty} \sum_{i=k}^{\infty} p^i = (1-p) \sum_{k=0}^{\infty} \left(p^k \sum_{i=0}^{\infty} p^i \right) = (1-p) \sum_{k=0}^{\infty} \frac{p^k}{1-p} = \sum_{k=0}^{\infty} p^k = \frac{1}{1-p}$$

Zauważmy, że oszacowanie to nie zależy od n i np. dla $p = \frac{1}{2}$ otrzymujemy średnio dwa wywołania rekurencyjne.

Możemy dalej oszacować liczbę wywołań rekurencyjnych na poziomie drugim:

$$n + \frac{n-1}{1-p} = n \left(1 + \frac{1}{1-p} \right) - \frac{1}{1-p} \leq n \frac{2-p}{1-p}$$

Biorąc pod uwagę wykładniczy rozrost drzewa przekłada się to na zmianę stopni wierzchołków w drzewie z n na $\sqrt{\frac{2-p}{1-p}}n$, co np. dla $p = \frac{1}{2}$ daje nam $\sqrt{3n}$.

Oczywiście powyższe oszacowanie jest zgrubne. Algorytm α - β obcięcia nigdy nie wykona więcej kroków niż algorytm mini-max, potrafi jednak istotnie zmniejszyć średnią liczbę wywołań rekurencyjnych, a co za tym zmieniać istotnie asymptotyczną złożoność programu.

Ćwiczenia

Rozwiązanie każdego z poniższych zadań wymaga użycia jednej z trzech technik: back-trackingu, programowania dynamicznego lub zachłannego. Pytanie której?

1. [PCh, Zadanie o misjonarzach i kanibalach] Przez rzekę chcą przepłynąć się kanibale i misjonarze. Kanibali i misjonarzy jest po trzech. Mają jedną łódkę, którą może płynąć do dwóch osób. Łódką umieją wiosłować kanibale i tylko jeden misjonarz. Jeżeli w dowolnej chwili, na dowolnym brzegu rzeki będzie więcej kanibali, niż misjonarzy, to zjedzą oni misjonarzy. W jaki sposób wszyscy mogą bezpiecznie przepłynąć się na drugi brzeg.
2. [PCh, Zadanie o trzech małżeństwach]. Przez rzekę chcą przepłynąć się trzy małżeństwa. Mają jedną łódkę, którą może płynąć do dwóch osób. Każda z kobiet może pozostać na brzegu w obecności mężczyzn tylko, jeżeli jest wśród nich jej mąż. W jaki sposób wszyscy mogą bezpiecznie przepłynąć się na drugi brzeg?
3. Na ile sposobów można wypełnić tablicę $n \times k$ liczbami ze zbioru $\{1, \dots, n \cdot k\}$, tak aby:
 - każda liczba pojawiała się dokładnie raz,
 - wiersze i kolumny tablicy były uporządkowane rosnąco.

Dla tablicy 3×3 jest ich 42. :-)

4. [CLR] Minimalny koszt mnożenia n macierzy, o wymiarach $x_1 \times x_2, x_2 \times x_3, \dots, x_n \times x_{n+1}$.
5. [CLR 16.3] Dane są dwa napisy (listy liter). Dostępne są operacje:
 - przepisz pierwszą literę z pierwszego ciągu na koniec drugiego,
 - zamień pierwszą literę pierwszego ciągu na inną literę i przenieś ją na koniec drugiego ciągu,
 - wstaw na koniec drugiego ciągu literę,
 - usuń z początku pierwszego ciągu pierwszą literę.

Każda z tych operacji ma dodatni koszt. Jaki jest minimalny koszt zamiany pierwszego ciągu w drugi?

6. [CLR 17.2-5] Dany jest zbiór punktów na osi. Podaj minimalną liczbę odcinków jednostkowych, jakimi można pokryć punkty z tego zbioru.
7. Rozważmy taką oto łamigłówkę: Dana jest lista liczb całkowitych $l = [x_1; x_2; \dots; x_n]$. W jednym ruchu możemy wybrać dowolne dwa równe elementy listy, usunąć je i wstawić na koniec listy ich sumę. Takie ruchy możemy powtarzać. Jaka jest minimalna długość listy jaką można uzyskać?

Napisz procedurę `lamiglowka : int list → int`, która dla danej listy l wyznaczy minimalną długość listy jaką można uzyskać.

Na przykład, `lamiglowka [4; 3; 4; 5; 12; 2; 3; 1; 6; 1] = 4`.

8. W korytarzu harcują myszy. Korytarz ma długość n metrów. Dana jest tablica n nieujemnych liczb całkowitych a opisująca gdzie jest ile myszy, dokładnie na odcinku od i do $i + 1$ metrów od wejścia do korytarza jest $a.(i)$ myszy.

Dysponujesz k kotami ($k \leq n$). Twoim zadaniem jest takie rozmieszczenie kotów w korytarzu, żeby złapały jak najwięcej myszy. Każdy kot może pilnować ustalonego przez Ciebie spójnego fragmentu korytarza (na przykład, może mieć założoną smycz odpowiedniej długości, przymocowaną do podłogi pośrodku pilnowanego fragmentu korytarza). Fragmenty korytarza pilnowane przez różne koty nie mogą zachodzić na siebie (żeby koty się nie pobiły, a smycze nie poplątały), choć mogą się stykać. Niektóre fragmenty korytarza mogą pozostać niepilnowane przez żadnego kota.

Kot, który pilnuje fragmentu korytarza od i do j metrów od wejścia (dla $i < j$), na którym znajduje się $s = a.(i) + a.(i+1) + \dots + a.(j-1)$ myszy, złapie: $\max(s - (j - i - 1)^2, 0)$ myszy.

Na przykład, dla $k = 2$ oraz $a = [1; 5; 1; 4; 3; 2; 7; 0]$ poprawnym wynikiem jest 14, jeden kot może pilnować fragmentu korytarza od 1 do 4 metrów od wejścia (łapiąc 6 z 10 myszy), a drugi może pilnować fragmentu korytarza od 4 do 7 metrów od wejścia (łapiąc 8 z 12 myszy).

Napisz procedurę `myszy : int → int array → int`, która dla danej liczby $k \geq 0$ oraz tablicy a , wyznaczy maksymalną liczbę myszy, jakie złapią koty.

Rozwiązanie tego zadania zostało omówione w *Delcie* z czerwca 2016 r.:

<http://www.deltami.edu.pl/temat/informatyka/algoritmy/2016/05/25/Myszy/>

9. Rozpatrujemy przedstawienia liczby naturalnej n jako sumy różnych składników nieparzystych, tzn. rozważamy takie ciągi (x_1, \dots, x_k) , że:

- $1 \leq x_i$ i x_i jest nieparzyste (dla $i = 1, \dots, k$),
- $x_1 > x_2 > \dots > x_k$,
- $x_1 + \dots + x_k = n$.

• Napisz procedurę, która dla danej liczby n obliczy, na ile różnych sposobów można ją przedstawić jako sumę różnych nieparzystych składników. Na przykład, dla $n = 2$ poprawnym wynikiem jest 0, a dla $n = 12$ poprawnym wynikiem jest 3.

• Napisz procedurę, która dla danej liczby n wyznaczy (pewne) przedstawienie tej liczby jako sumy maksymalnej liczby nieparzystych składników. Na przykład, dla $n = 42$ poprawnym wynikiem może być $[15; 11; 7; 5; 3; 1]$.

10. Rozważamy ciągi liczb całkowitych (x_1, x_2, \dots, x_k) konstruowane zgodnie z następującymi regułami:

- $x_1 = 1$,
- $x_{i+1} = x_i + 1$, lub
- $x_{i+1} = x_i \cdot f_i$ dla pewnej liczby Fibonacciego f_i .

Napisz procedurę `ciag : int → int list`, która dla danej dodatniej liczby całkowitej n wyznaczy najkrótszy taki ciąg (x_1, x_2, \dots, x_k) , że $x_k = n$.

Na przykład `ciag 42` powinno zwrócić $[1; 2; 42]$ lub $[1; 21; 42]$.

11. Tworzymy ciąg liczb całkowitych (x_k) w następujący sposób:

- $x_0 = 1$,
- dla każdego $k > 0$ wybieramy pewne $0 \leq i, j < k$ i przyjmujemy $x_k = x_i + x_j$.

Napisz procedurę `ciąg: int → int list`, która dla danego $n \geq 1$ znajdzie **najkrótszy** taki ciąg x_0, x_1, \dots, x_k , który spełnia powyższe warunki, oraz $x_k = n$. Wynikiem procedury powinna być lista $[x_0; \dots; x_k]$.

Rozwiązując to zadanie można łatwo wybrać złą technikę. Konieczne jest więc solidne uzasadnienie poprawności. Oto garść kontrprzykładów, które należy wziąć pod uwagę:

n	Wynik
15	1, 2, 3, 6, 12, 15
382	1, 2, 4, 5, 9, 14, 23, 46, 92, 184, 198, 382
77	1, 2, 4, 5, 9, 18, 36, 41, 77
12509	1, 2, 3, 6, 12, 13, 24, 48, 96, 192, 384, 397, 781, 1562, 3124, 6248, 6261, 12509

12. Rozważamy ciągi liczb całkowitych (x_1, x_2, \dots, x_k) konstruowane zgodnie z następującymi regułami:

- $x_1 = 1$,
- $x_{i+1} = x_i + 1$, lub
- $x_{i+1} = x_i \cdot p_i$ dla pewnej liczby pierwszej p_i .

Napisz procedurę `ciąg: int → int list`, która dla danej dodatniej liczby całkowitej n wyznaczy najkrótszy taki ciąg (x_1, x_2, \dots, x_k) , że $x_k = n$.

Możesz założyć, że dane są procedury:

- `prime: int → bool` sprawdzająca czy dana liczba jest pierwsza, oraz
- `primes: int → int list`, która dla danej dodatniej liczby całkowitej n zwraca listę liczb pierwszych nieprzekraczających n (uporządkowaną rosnąco).

13. Dla danych dodatnich liczb całkowitych l i m szukamy przedstawienia ułamka $\frac{l}{m}$, jako możliwie najkrótszej sumy odwrotności dodatnich liczb całkowitych:

$$\frac{l}{m} = \frac{1}{s_1} + \frac{1}{s_2} + \dots + \frac{1}{s_k}$$

Napisz procedurę `rozklad: int*int → int list`, która dla danych liczb l i m wyznaczy szukany ciąg mianowników $[s_1; \dots; s_k]$.

14. Dana jest dodatnia liczba całkowita x . Interesują nas różne rozkłady x na sumę różnych (dodatnich) liczb Fibonacciego. Dwa rozkłady, które różnią się jedynie kolejnością składników, uznajemy za takie same. Tylko rozkłady, które mają inne zbiory składników, uznajemy za różne.

Napisz procedurę `rozklady: int → int`, która dla danej liczby x policzy na ile różnych sposobów można przedstawić x jako sumę różnych (dodatnich) liczb Fibonacciego.

Na przykład `rozklady 42 = 6`.

15. Dana jest lista dodatnich liczb całkowitych $[x_1; x_2; \dots; x_n]$ oraz dodatnia liczba całkowita k . Napisz procedurę `podzial : int list → int → int list list`, która podzieli daną listę na k spójnych fragmentów $[[x_1; \dots; x_{n_1}]; [x_{n_1+1}; \dots; x_{n_2}]; \dots; [x_{n_{k-1}+1}; \dots; x_n]]$ w taki sposób, że $\max\{x_1 + \dots + x_{n_1}, x_{n_1+1} + \dots + x_{n_2}, \dots, x_{n_{k-1}+1} + \dots + x_n\}$ jest jak najmniejsze. Jeżeli $k > n$, to w liście wynikowej mogą występować puste listy.

Na przykład `podzial [1; 4; 2; 2; 4; 1; 2; 4] 3 = [[1; 4; 2]; [2; 4]; [1; 2; 4]]`.

16. Ciąg $[x_1; x_2; \dots]$ nazwiemy *naprzemiennym*, jeżeli:

- $x_1 \neq x_2$,
- $x_1 = x_3 = x_5 = \dots$, oraz
- $x_2 = x_4 = x_6 = \dots$.

Ciąg jednoelementowy i pusty są również naprzemienne.

Napisz procedurę `naprzemienny : int array → int`, która mając dany ciąg liczb całkowitych w postaci tablicy obliczy długość najdłuższego jego naprzemiennego podciągu.

Na przykład, `naprzemienny [|0;4;7;4;1;2;4;0;2;5;4;2;7;2;1;3;4|] = 7`.

17. [XVI OI, zadanie Słonie] Dane jest tablica p długości n zawierająca permutację liczb ze zbioru $\{0, \dots, n-1\}$. Chcemy posortować tablicę p rosnąco, ale jedyną operacją modyfikującą, którą możemy wykonywać na niej jest zamiana zawartościami dwóch komórek o podanych indeksach. Zamiana komórek zawierających liczby x i y kosztuje $x + y + 1$. Napisz procedurę `koszt : int array → int`, która oblicza minimalny koszt posortowania tablicy p .

18. [Bentley, CLR 16-1] Bitoniczne rozwiązanie problemu komiwojażera to taki cykl w kształcie wielokąta, który począwszy od skrajnie lewego wierzchołka, idzie w prawo (tworząc górny obrys wielokąta) do skrajnie prawego, po czym wraca do skrajnie lewego (tworząc obrys dolny). Wyznacz najkrótszy bitoniczny cykl Hamiltona dla danego zbioru punktów na płaszczyźnie.

19. [II OI] Dany jest zestaw n czynności. Wykonanie każdej z tych czynności trwa tym dłużej, im później się ją zacznie, konkretnie jeżeli zaczniemy wykonywać czynność i w chwili t , to jej wykonanie będzie trwać $a_i t + b_i$. Jaki jest minimalny czas wykonania wszystkich tych czynności, jeżeli zaczynamy je wykonywać w chwili 0?

20. Rozważamy drzewa postaci:

```
type expr =
  NWD of expr * expr |
  NWW of expr * expr |
  Number of int
```

Drzewa takie reprezentują wyrażenia. NWD oznacza największy wspólny dzielnik, a NWW najmniejszą wspólną wielokrotność dwóch podwyrażeń. Argumentami `Number` są liczby nieujemne. `Number x`, dla $x > 0$ oznacza liczbę x . `Number 0` oznacza miejsce, w które należy wstawić dodatnią liczbę.

Napisz procedurę `wstaw : expr * int → int`, która dla danego drzewa t i dodatniej liczby całkowitej n znajdzie taką najmniejszą dodatnią liczbę całkowitą x , że gdy wstawimy `Number x` w miejsce **wszystkich** wystąpień `Number 0` w t , to wartością wyrażenia będzie n . Jeżeli taka liczba x nie istnieje, to poprawnym wynikiem jest 0.

Na przykład, dla $t = \text{NWW}(\text{NWW}(\text{Number } 2, \text{Number } 0), \text{NWD}(\text{Number } 0, \text{Number } 49))$ i $n = 42$, `wstaw t n = 21`.

21. Dane jest drzewo typu:

```
type choinka =  
  Koniuszek |  
  Galaz of choinka |  
  Rozgalezienie of choinka * choinka;;
```

opisujące kształt choinki.

- (a) [PCh] W węzłach choinki (`Rozgalezienie`, `Galaz` i `Koniuszek`) możemy umieszczać świecek. Powiemy, że choinka jest oświetlona, jeżeli dla każdej krawędzi (łączyjącej węzły) choinki przynajmniej w jednym jej końcu znajduje się świeczka. Napisz procedurę, która dla danej choinki obliczy minimalną liczbę świeczek potrzebnych do oświetlenia choinki.
- (b) [PCh] Na każdej krawędzi (łączyjącej węzły) choinki wieszamy jedną bombkę. Mamy do dyspozycji trzy rodzaje bombek: czerwone po 2 zł, niebieskie po 5 zł i srebrne po 7 zł. Choinkę należy przystroić w taki sposób, żeby na krawędziach o końcach w tym samym węźle wisały bombki o różnych kolorach, a łączny koszt przystrojenia całej choinki był jak najmniejszy.
- (c) W węzłach choinki (`Rozgalezienie`, `Galaz` i `Koniuszek`) wieszamy bombki. Dostępne są trzy rodzaje bombek:
 - czerwone, wykonane z miedzi, każda taka bombka waży 1 kg,
 - srebrne, wykonane z białego złota niskiej próby, każda taka bombka waży 2 kg,
 - złote, wykonane ze szczyrego złota, każda taka bombka waży 3 kg.

Bombki należy zawiesić tak, aby choinka nie chwiała się, tzn. dla każdego rozgałęzienia łączny ciężar bombek zawieszonych po lewej i po prawej stronie musi być taki sam. Pytanie czy jest to możliwe, a jeżeli tak, to na ile sposobów?

Napisz procedurę, która dla danej choinki obliczy na ile sposobów można powiesić na niej bombki.

Wytyczne dla prowadzących ćwiczenia

- Zadanie o drzewach NWW i NWD można rozwiązać tak:

```
let wstaw expr n =  
  let f x = (* wylicz expr z wstawionym x *) in  
    if f n <>n then 0  
    else  
      let rec licz d =  
        if gcd d n = gcd (d * d) n then gcd d n  
        else licz (d * d)  
      in licz (n / f 1)
```

Rozwiązanie, które rozkłada n na liczby pierwsze i rozpatruje każdą liczbę pierwszą osobno jest trochę wolniejsze.

Wykład 23. Gry typu wygrana/przegrana i drzewa AND-OR

23.1 Drzewa AND-OR

[[Przerobić. W DAG-u trzeba uaktualniać wszystkie ścieżki prowadzące do zmodyfikowanego liścia.]]

W niniejszym wykładzie przedstawimy materiał dotyczący gier kończących się wynikiem binarnym — wygrana/przegrana. W przypadku takich gier wagi przyporządkowywane węzłom drzewa mini-max mogą przyjmować tylko dwie wartości. Jeśli przyjmiemy, że naszą wygraną oznaczymy przez `true`, a przeciwnika przez `false`, to zamiast mówić o maksimum i minimum możemy powiedzieć, że my gramy na `or`, a przeciwnik na `and` — stąd nazwa: drzewa AND-OR.

Rozpatrujemy częściowo rozwinięte drzewo gry. Wierzchołki drzewa są typu AND lub OR, w zależności od tego, kto w danej sytuacji ma wykonać ruch. Liście mogą reprezentować sytuacje końcowe lub sytuacje, w których gra może się dalej rozwijać. Liście reprezentujące sytuacje końcowe mają przyporządkowane wagi zgodnie z tym, kto wygrał. Liście reprezentujące sytuacje nierozstrzygnięte mają wagi nieokreślone. Węzły wewnętrzne mają określone wagi, jeśli można je określić na podstawie wag ich potomków.

Jeśli korzeń drzewa ma wyznaczoną wagę, to określa ona, czy mamy strategię wygrywającą, a wagi synów korzenia wskazują, jaki ruch należy wykonać. Jeżeli korzeń drzewa nie ma określonej wagi, to należy rozwinąć drzewo. Pytanie tylko w którym miejscu?

23.2 Zbiory falsyfikujące/potwierdzające

Załóżmy, że korzeń ma wagę nieokreśloną.

Fakt: Nadając liściom z nieokreślonymi wagami wagi `true` możemy doprowadzić do tego, że korzeń ma wagę `true`, i odwrotnie, nadając liściom wagi `false` możemy doprowadzić do tego, że korzeń będzie miał wagę `false`.

Dowód: Bo operacje `and` i `or` są monotoniczne.

Definicja: Zbiór *falsyfikujący* (*potwierdzający*) to taki zbiór liści o nieokreślonych wagach, który jeżeli liście te uzyskają wagi `false` (`true`), to korzeń uzyska też wagę `false` (`true`).

Fakt: Dowolne dwa zbiory potwierdzający i falsyfikujący mają niepuste przecięcie.

Dowód: Gdyby tak nie było, to moglibyśmy przypisać liściom z jednego zbioru wagi `true`, a z drugiego `false` i korzeniowi można by przypisać równocześnie obie wagi — sprzeczność.

Interesują nas **minimalne** zbiory falsyfikujące i potwierdzające. Jeżeli określimy wartość wierzchołka należącego do przecięcia zbioru falsyfikującego i potwierdzającego, to wyeliminujemy jeden z nich. Strategia polega na tym, aby cyklicznie rozwijać wierzchołek, który należy do przecięcia zbiorów falsyfikującego i potwierdzającego o minimalnej liczbie elementów. Szukany wierzchołek możemy wyznaczyć obchodząc drzewo i korzystając z następujących faktów:

Fakt:

- Jeżeli korzeń drzewa jest typu **and**, to:
 - najmniejszy zbiór falsyfikujący to zbiór falsyfikujący dla jednego z synów,
 - najmniejszy zbiór potwierdzający to suma najmniejszych zbiorów potwierdzających dla synów,
 - szukany element jest w tym poddrzewie, które ma najmniejszy zbiór falsyfikujący.
- Jeżeli korzeń drzewa jest typu **or**, to dualnie.

Rozwijamy tak drzewo gry, aż określimy wartość korzenia.

23.3 Zwinięcie drzewa do DAG-u

Opisany algorytm możemy usprawnić, nie konstruując drzewa, lecz sklejjąc wierzchołki odpowiadające identycznym sytuacjom na planszy. W ten sposób uzyskujemy DAG. Problem wyznaczenia najmniejszego zbioru falsyfikującego/potwierdzającego dla DAG-u jest trudny. Stosuje się więc następującą heurystykę: wybieramy do rozwinięcia takie wierzchołki, jakbyśmy mieli drzewo stanowiące rozwinięcie DAGu. Oczywiście rozwijając DAG, należy pilnować, aby sklejać powtarzające się wierzchołki. Możemy tu stosować spamiętywanie sytuacji, dla których utworzyliśmy wierzchołki.

23.4 Implementacja

Definicje wstępne, niezależne od gry:

```
type player = And | Or;;
let i = Or;;

let other = function And ->Or | Or ->And;;

type boolean = True | False | Undefined;;

let and3 x y =
  match (x,y) with
  (False,_) ->False |
  (_,False) ->False |
  (True,_) ->y |
  (_,True) ->x |
  _ ->Undefined;;

let or3 x y =
  match (x,y) with
  (True,_) ->True |
  (_,True) ->True |
  (False,_) ->y |
  (_,False) ->x |
  _ ->Undefined;;

let won = True;;
let lost = False;;
```

Zakładamy, że gra jest określona podobnie, jak gra w kamienie w poprzednich wykładach.

```
(* Drzewo AND-OR jest reprezentowane za pomocą mapy przyporządkowującej *)
(* parom (sytuacja,gracz) n-tki postaci: *)
(* - wartość: True / False / Undefined *)
(* - minimalna liczba el. falsyfikujących, *)
(* - minimalna liczba el. potwierdzających, *)
(* - dla rozwiniętych: lista trójek (ruch,sytuacja,gracz). *)
```

```
(* Selektory *)
```

```
let vertex dag s p = apply dag (s,p);;
```

```
let value (v,_,_,_) = v;;
let min_f (_,f,_,_) = f;;
let min_t (_,_,t,_) = t;;
let succ (_,_,_,s) = s;;
```

```
let succ_move (m,_,_) = m;;
let succ_board (_,b,_) = b;;
let succ_who (_,_,who) = who;;
let follow dag x = vertex dag (succ_board x) (succ_who x);;
```

```
(* Węzeł jest rozwinięty, jeżeli ma synów lub ma wartość. *)
let is_expanded v = (value v <>Undefined) || (succ v <>[]);;
```

```
(* Operacje na DAG-u *)
```

```
(* Czy już jest taki wierzchołek? *)
```

```
let is_present dag s who = dom dag (s,who);;
```

```
(* Dodanie do DAG-u węzła o parametrach: *)
(* - s - sytuacja, *)
(* - who - gracz, *)
(* - v - wartosc, *)
(* - e - lista trójek (ruch,sytuacja,gracz). *)
let insert dag s who v e =
```

```
(* Rozmiar najmniejszego zbioru falsyfikującego/potwierdzającego *)
(* (w zależności od parametrów selector/cumulator) dla synów. *)
```

```
let minft selector cumulator =
```

```
  (* Lista liczb do przeliczenia, pochodzących od *)
  (* synów z nieokreślonymi wartościami. *)
  let minl = map selector
    (filter
      (fun x ->value x = Undefined)
      (map (follow dag) e))
```

```
  in
```

```
    if e = [] && v = Undefined then
      (* Wierzchołek nierozwinięty. *)
```

```
    1
```

```

else if minl <>[] then
  (* Zliczenie ilości wierzchołków wyznaczających wartość. *)
  cumulator minl
else
  (* Wszyscy synowie są określani. *)
  infty

(* Przelicz wartość wierzchołka. *)
and recalculate_value =
  if v = Undefined && e <>[] then
    let ss = map (fun x ->value (follow dag x)) e
    in
      if who = And then
        fold_left and3 True ss
      else
        fold_left or3 False ss
    else v
  in
    (* Liczba wierzchołków falsyfikujących. *)
    let minf = minft min_f (if who = Or then sum else minl)

    (* Liczba wierzchołków potwierdzających. *)
    and mint = minft min_t (if who = And then sum else minl)
    in

      update dag (s,who) (recalculate_value,minf,mint,e);;

(* Dodanie nierozwiniętego liścia. *)
let insert_leaf dag s who =
  if is_present dag s who then
    dag
  else
    insert dag s who Undefined [];;

(* Początkowa sytuacja do rozważenia. *)
let initial s = insert_leaf empty s i;;

(* Rozwija w danym dag-u węzeł v odpowiadający sytuacji s dla gracza who. *)
let rec expand_dag dag s who =

  (* Rozwija liść v odpowiadający sytuacji s. *)
  let expand_leaf v =
    if finished s who then begin
      (* Rozgrywka skończona. *)
      insert dag s who (result s who) []
    end else
      (* Rozwiń możliwe ruchy. *)
      let leafs = map (fun m ->(m,move s who m)) (moves s who)
      in
        (* Wstaw nowych synów do DAG-u. *)
        let dag1 =

```

```

        fold_left
          (fun d (m,s) ->insert_leaf d s (other who))
            dag leafs
      (* Lista następników. *)
      and e = map (fun (m,s) ->(m,s,other who)) leafs
    in
      insert dag1 s who (value v) e

(* Przelicza wartość węzła na podstawie wartości potomków. *)
and recalculate_value d v =
  let ss = map (fun x ->value (follow d x)) (succ v)
  in
    let vv =
      if who = And then
        fold_left and3 True ss
      else
        fold_left or3 False ss
    in
      insert d s who vv (succ v)
in

(* Rozwija podgraf zaczepiony w węźle v wybierając minimalną *)
(* wartość podanego aspektu. *)
let expand_compound v aspect =
  (* Predykat sprawdzający, czy syn wymaga rozwinięcia. *)
  let test s =
    let vv = follow dag s
    in aspect vv = aspect v && value vv = Undefined
  in
    (* Rozwijamy syna realizującego min-f/min-t *)
    let ssl = filter test (succ v)
    in
      if ssl = [] then
        (* Potomek jest już rozwinięty. *)
        recalculate_value dag v
      else
        let dag1 = expand_dag dag (succ_board (hd ssl)) (other who)
        in
          recalculate_value dag1 v
in
  let v = vertex dag s who
  in
    if not (is_expanded v) then
      (* Rozwinięcie liścia. *)
      expand_leaf v
    else if succ v = [] then
      (* Nie ma co rozwijać. *)
      dag
    else if who = And then
      (* Rozwinięcie węzła And. *)
      expand_compound v min_f
    else
      (* Rozwinięcie węzła Or. *)

```

```

    expand_compound v min_t;;

(* Rozwijaj, aż zostanie określona waga korzenia. *)
let rec keep_expanding dag s who =
  if value (vertex dag s who) = Undefined then
    keep_expanding (expand_dag dag s who) s who
  else
    dag;;

(* Najlepszy ruch. *)
let game s =
  let dag = initial s
  in
  let dag1 = keep_expanding dag s i
  in
  let v = vertex dag1 s i
  in
  (succ_move (hd (filter
    (fun s ->value v = value (follow dag1 s))
    (succ v)))));;

```

Wykład 24. Wyszukiwanie wzorców

Zastanówmy się nad następującym problemem. Mamy dane dwa teksty, szukamy wystąpień pierwszego tekstu w drugim. Na nasze potrzeby przyjmujemy, że tekst to lista symboli. Wynikiem powinna być lista pozycji wszystkich wystąpień wzorca w tekście.

24.1 Naiwny algorytm wyszukiwania wzorca

Algorytm naiwny sprawdza wszystkie możliwe pozycje, na których wzorec występuje w tekście i sprawdza, czy faktycznie tak jest.

```
let rec prefix w t =
  match w with
  [] ->true |
  hw::tw ->
    match t with
    [] ->false |
    ht::tt ->(hw = ht) && prefix tw tt;;

let pattern_match w t =
  let rec iter t n acc =
    let a = if prefix w t then n::acc else acc
    in
    match t with
    [] ->rev a |
    _::tail ->iter tail (n+1) a
  in
  iter t 1 [];;
```

Złożoność czasowa tego algorytmu jest rzędu $\Theta(|w| \cdot |t|)$. Czy można lepiej? Oczywiście tak.

24.2 Algorytm Rabina-Karpa

[[Uzupełnić, CLR p.34.2]]

24.3 Algorytm MP

Algorytm przedstawiony poniżej pozwala wyznaczyć wszystkie wystąpienia wzorca w tekście w czasie liniowym. Jego zastosowanie jest jednak trochę szersze. Dokładniej, algorytm ten dla danego tekstu wyznacza wartości tzw. funkcji prefiksowej P . Dla danego tekstu $t = (t_1 t_2 \dots t_n)$ funkcja prefiksowa jest zdefiniowana następująco.

$P(i)$ to największe takie j , że:

- $0 \leq j < i$,
- $(t_1 \dots t_j) = (t_{i-j+1} \dots t_i)$.

Inaczej mówiąc, $P(i)$ to długość najdłuższego właściwego prefiksu $(t_1 t_2 \dots t_i)$, który jest równocześnie sufiksem $(t_1 t_2 \dots t_i)$.

Oznaczmy dodatkowo przez $G(i)$ zbiór

$$G(i) = \{0 \leq j < i : (t_1 \dots t_j) = (t_{i-j+1} \dots t_i)\}$$

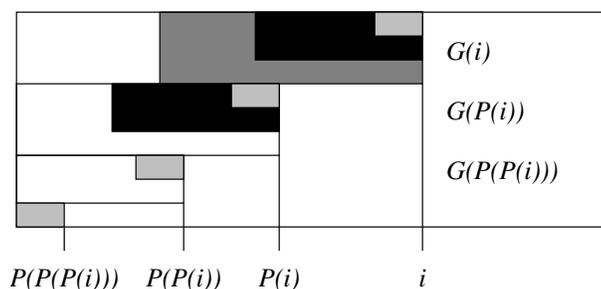
Czyli $G(i)$ to zbiór długości wszystkich takich właściwych prefiksów $(t_1 t_2 \dots t_i)$, które są równocześnie sufiksami tego słowa. Dodatkowo przyjmujemy, że:

$$G(0) = \emptyset$$

$$P(0) = 0$$

Zachodzą następujące własności:

- $P(i) = \max G(i)$,
- $P(1) = 0$,
- $P(i) \geq 0$,
- $P(i) < i$ (dla $i > 0$),
- $i > P(i) > P(P(i)) > \dots > P^k(i) = 0$, dla pewnego $k \in \mathbb{N}$,
- $G(i+1) = \{0\} \cup \{j+1 : j \in G(i) \wedge t_{i+1} = t_{j+1}\}$,
- $G(i) = \{P(i)\} \cup G(P(i))$ — największy element w $G(i)$ to właśnie $P(i)$, natomiast każdy mniejszy element $G(i)$ to długość prefiksu tekstu, który jest sufiksem $(t_1 t_2 \dots t_{P(i)})$.



- $G(i) = \{P^j(i) : 1 \leq j \leq k\}$,
- $G(i+1) = \{0\} \cup \{P^j(i) + 1 : 1 \leq j \leq k \wedge t_{i+1} = t_{P^j(i)+1}\}$,
- $P(i+1) = \max(\{0\} \cup \{P^j(i) + 1 : 1 \leq j \leq k \wedge t_{i+1} = t_{P^j(i)+1}\})$,
- dla $i > 0$ mamy $P(i+1) = \begin{cases} P^j(i) + 1 & ; \text{gdzie } j \text{ jest najmniejsze takie, że} \\ & 1 \leq j \leq k \wedge t_{i+1} = t_{P^j(i)+1} \\ 0 & ; \text{wpp.} \end{cases}$,

A oto i implementacja:

```
let pref t =
  let
    p = make (length(t) + 1) 0 and
    pj = ref 0
  in begin
    for i = 2 to length t do
      while (!pj > 0) && (t.(!pj) <> t.(i - 1)) do
        pj := p.(!pj)
      done;
      if t.(!pj) = t.(i - 1) then pj := !pj + 1;
      p.(i) <- !pj
    done;
  p
end;;
```

Chcąc zastosować algorytm MP do wyszukiwania wzorców, obliczamy tablicę prefiksową dla szukanego wzorca. Z jej pomocą możemy znaleźć wystąpienia wzorca w czasie liniowym.

```
let find x y =
  let
    i = ref 0 and
    j = ref 0 and
    w = ref [] and
    p = pref x
  in
    while !i <= length y - length x do
      j := p.(!j);
      while (!j < length x) && (x.(!j) = y.(!i + !j)) do
        j := !j + 1
      done;
      if !j = length x then w := !i::!w;
      i := !i + if !j > 0 then !j - p.(!j) else 1
    done;
  rev !w;;
```

```
find [|'a'; 'l'; 'a'|] [|'a'; 'l'; 'a'; 'l'; 'a'; 'b'; 'a'; 'm'; 'a'|];;
[0; 2]
```

24.4 Zastosowanie tablicy prefiksowej w automatach skończonych

[[Uzupełnić]]

[[Zastanović się nad dodaniem algorytmu Boyera-Moora.]]

Ćwiczenia

1. Dana jest lista elementów. Wyznacz wszystkie nietrywialne rotacje cykliczne danej listy, które dają w wyniku ją samą.
2. Dane są dwa napisy (w postaci tablic znaków) x i y . Napisz procedurę `szukaj` : `char array` \rightarrow `char array` \rightarrow `int`, która znajduje najdłuższy spójny fragment w napisie x , który jest prefiksem (fragmentem początkowym) napisu y . Wynikiem procedury powinna być jego długość.
3. Okresem ciągu $[x_1; x_2; \dots; x_n]$ nazwiemy najmniejszą dodatnią liczbę całkowitą k , taką że $x_i = x_{i+k}$, dla $i = 1, 2, \dots, n - k$. Napisz procedurę `okresy` : `int list` \rightarrow `int list`, której wynikiem dla danej listy $[x_1; x_2; \dots; x_n]$ jest taka lista $[y_1; y_2; \dots; y_n]$, że y_i jest okresem $[x_i; x_{i+1}; \dots; x_n]$.
4. Niech (x_1, x_2, \dots, x_n) będzie danym niepustym ciągiem liczb. Okresem całkowitym tego ciągu nazwiemy najmniejszą taką liczbę k , że:
 - $1 \leq k \leq n$,
 - k jest dzielnikiem n ,
 - $x_i = x_{i+k}$ dla wszystkich $i = 1, \dots, n - k$.

Zauważmy, że każdy ciąg ma okres całkowity, co najwyżej jest to $k = n$. Na przykład, okresem całkowitym ciągu $[1; 3; 2; 1; 1; 3; 2; 1; 1; 3; 2; 1]$ jest 4, a okresem całkowitym ciągu $[1; 3; 2; 1; 1]$ jest 5.

Napisz procedurę `okres`: `int array` \rightarrow `int`, która dla danego ciągu wyznaczy jego okres całkowity.

5. Dana jest tablica a zawierająca ciąg n elementów. Napisz procedurę `prepalindrom` : `alpha array` \rightarrow `int`, która dla danej tablicy a znajduje długość najdłuższego jej prefiksu, który jest palindromem.
6. Dana jest tablica a zawierająca ciąg n elementów. Napisz procedurę `presuf` : `alpha array` \rightarrow `int`, która dla danej tablicy a znajduje długość najdłuższego jej prefikso-sufiksu, który ma przynajmniej trzy rozłączne wystąpienia w a .
7. Dana jest tablica a zawierająca ciąg n elementów. Napisz procedurę `double` : `alpha array` \rightarrow `int`, która dla danej tablicy a znajduje długość i najdłuższego jej prefiksu, który powtórzony dwukrotnie nadal jest jej prefiksem, tzn. $a.(0) = a.(i)$, $a.(1) = a.(i + 1)$, \dots , $a.(i - 1) = a.(2i - 1)$.
8. [XII OI, zadanie *Szablon*] Chcemy wykonać długi napis. W tym celu najpierw przygotowujemy odpowiedni szablon z wyciętymi literkami. Następnie taki szablon można przyłożyć we właściwym miejscu do ściany i malujemy po nim farbą. Malujemy zawsze po całym szablonie, ale dopuszczamy, aby litery były malowane wielokrotnie. Literki na szablonie znajdują się koło siebie (nie ma tam przerw). Zadanie polega na wyznaczeniu minimalnej długości szablonu, za pomocą którego można wykonać cały napis.
9. [XIII OI, zadanie *Okresy słów*] Napis Q jest okresem A , jeśli Q jest prefiksem właściwym A oraz A jest prefiksem (niekoniecznie właściwym) napisu QQ . Przykładowo, napisy

abab i *ababab* są okresami napisu *abababa*. Maksymalnym okresem napisu A nazywamy najdłuższy z jego okresów, lub napis pusty, jeśli A nie posiada okresu. Dla przykładu, maksymalnym okresem napisu *ababab* jest *abab*. Maksymalnym okresem *abc* jest napis pusty.

Napisz procedurę, która dla danego (w postaci listy) napisu obliczy listę długości maksymalnych okresów kolejnych jego prefiksów.

Wytyczne dla prowadzących ćwiczenia

Ad. 1 Rozwiązanie: wyszukujemy danego tekstu w w tekście ww .

Ad. 3 Najpierw odwracamy listę — sufiksy zamieniają się na prefiksy. Następnie liczymy tablicę prefiksową P . Okres prefiksu do pozycji i , to $i - P[i]$.

Ad. 8 Rozwiązanie musi być prefikso-sufiksem danego tekstu. Wszystkie prefikso-sufiksy możemy wyznaczyć korzystając z tablicy prefiksowej i algorytmu [K]MP. Nie musimy jednak sprawdzać wszystkich z nich. Jeżeli kolejne prefikso-sufiksy mają długości k i l oraz $\frac{l}{2} \leq k < l$, to nie musimy sprawdzać, czy prefikso-sufiks długości l jest szablonem – nawet jeżeli tak, to prefikso-sufiks długości k jest lepszym szablonem.

Wykład 25. Tablica sufiksowa

Konstrukcja tablicy sufiksowej

Wyszukiwanie wielu wzorców

Ćwiczenia

1. Dana jest tablica a zawierająca n znaków. Rotacją cykliczną takiej tablicy o i (dla $0 \leq i < n$) nazwiemy tablicę powstałą z przeniesienia elementów $a.(0), \dots, a.(i-1)$ z początku tablicy na jej koniec, przy równoczesnym przesunięciu elementów $a.(i), \dots, a.(n-1)$ o i pozycji w lewo.

Napisz procedurę `minrot : char array → int`, która dla danej tablicy znaków a zwróci taką liczbę i , że rotacja tablicy a o i pozycji jest najmniejsza w porządku leksykograficznym, spośród wszystkich jej rotacji cyklicznych.

2. Dane są dwa napisy (w postaci tablic znaków) x i y . Napisz procedurę `szukaj : char array → char array → int * int`, która znajduje najdłuższy spójny fragment występujący w obu napisach. Wynikiem procedury powinna być para: pozycje jego początków w obu napisach.
3. Dany jest tekst w postaci tablicy znaków $[[c_1; \dots; c_n]]$. Napisz procedurę `podślowo : char array → (int * int)`, która znajduje takie pozycje w tekście (i, j) , że:

- $0 < i \leq j \leq n$,
- liczba wystąpień k napisu $[[c_i; \dots; c_j]]$ w danym tekście, pomnożona przez jego długość (czyli $k \cdot (j - i + 1)$) jest maksymalna.

Dla pustej tablicy poprawnym wynikiem jest $(0, 0)$.

Wykład 26. Procedury dużo wyższych rzędów

Jaka jest różnica między danymi i procedurami? W programowaniu funkcyjnym to rozróżnienie rozmywa się. Dane mogą być traktowane jak procedury — każdy interpreter czy kompilator zamienia dane (kod źródłowy programu) na procedurę (wykonywalny program). Procedury wyższych rzędów operują na innych procedurach jak na danych. Tak więc procedury mogą być również danymi. Można by powiedzieć, że dane tym różnią się od procedur, że zawsze są podmiotem obliczeń, a nigdy nie są wykonywane. Niniejszy wykład powinien Państwa przekonać, że wcale tak nie musi być. Używając języka programowania nie widzimy, w jaki sposób zrealizowane są struktury danych i na jak jest zrealizowane wykonywanie procedur. Łatwo sobie wyobrazić, że procedury mogą mieć postać kodu źródłowego lub częściowo skompilowanego, który jest *interpretowany*. Tak więc procedury mogą być zrealizowane w postaci danych (interpretowanych przez procedurę ewaluatora).

Podobnie, dane mogą być procedurami. Nie widzimy sposobu implementacji podstawowych struktur danych wbudowanych w język programowania. Poznamy teraz jedną z możliwych ich implementacji — całkowicie proceduralną.

Disclaimer: Niniejszy wykład ma charakter ćwiczenia umysłowego. Struktury danych wcale nie są implementowane w opisany sposób, ani nie jest to najlepszy sposób ich implementacji. Jest to jednak możliwe. Celem tego ćwiczenia jest zilustrowanie w pełni zamiennego traktowania danych jak procedur i procedur jak danych. Zdobywszy tę umiejętność, będziemy mogli wznieść się na wyższy poziom abstrakcji i tworzyć struktury, w których dane są przemieszane z procedurami.

Niektóre programy, ze względu na system typów Ocamlu działają „jednorazowo”, tzn. każda konstrukcja jest poprawna i została przetestowana, jednak użycie jednej konstrukcji może uniemożliwić użycie innej.

26.1 Definiowanie form specjalnych

W Ocamlu argumenty procedur są obliczane *gorliwie*, tzn. najpierw są obliczane ich wartości, a dopiero potem przystępujemy do obliczania wartości procedury. Wyjątkiem są *formy specjalne*, np. `if` — w zależności od wartości warunku, tylko jeden z pozostałych członów jest obliczany. Czy można takie formy zaimplementować samemu? Tak. Potrzeba do tego dwóch rzeczy. Musimy umieć zabezpieczać wartości przed zbyt wczesnym obliczaniem i definiować *makrodefinicje*.

26.1.1 Makrodefinicje

W Ocamlu możemy zmieniać składnię języka, wprowadzając własne makrodefinicje. Służy do tego preprocesor P4. Nie będziemy tu mówić o nim, lecz sporadycznie użyjemy definicji wprowadzających potrzebne nam formy specjalne.

26.2 Uleniwianie

Model obliczeniowy Ocamlu charakteryzuje się zachłannym obliczaniem wartości — argumenty funkcji są obliczane bez względu na to, czy są potrzebne. Uleniwianie polega na zabezpieczeniu wartości przed obliczeniem, jeżeli nie jest to potrzebne. Zamiast wartości mamy „obietnicę jej obliczenia”, czyli procedurę bezargumentową, której wartością jest dana wartość. W momencie

gdy wartość jest nam potrzebna, obliczamy ją. Żeby nie obliczać tej wartości wielokrotnie, stosujemy spamiętywanie. W Ocamlu dostępna jest forma specjalna `lazy` i procedura `force`. `Lazy` powoduje odroczenie (ze spamiętywaniem) obliczenia wyrażenia. Chcąc obliczyć wartość wykonujemy na wyniku `lazy` operację `force`. Prostsza wersja uleniwiania, bez spamiętywania, może wyglądać tak:

```
'a lazy_t ≡ unit ->'a
lazy w ≡ function () ->w
let force x = x ();;
let a = lazy 4;;
force a;;
```

Pełne uleniwianie zawiera również spamiętywanie:

```
type 'a value = NoValue | Exception of exn | Value of 'a;;

let memoize f =
  let
    v = ref NoValue
  in
    function () ->
      match !v with
      | Value y ->y |
      | Exception e ->raise e |
      | NoValue ->
          let
            y = try f ()
                with e ->begin
                  v := Exception e;
                  raise e
                end
          in
            v := Value y;
            y;;

lazy w ≡ memoize (fun () ->w)

let b = lazy (print_int 42; 42);;
force b;;
force b;;
```

26.3 Wartości logiczne

Aby mówić o wartościach logicznych potrzebujemy:

- prawdę i fałsz,
- (`if ...`),
- koniunkcję, alternatywę i negację.

Wartość logiczną reprezentuję jako procedurę dwuargumentową, która w przypadku prawdy wybiera pierwszy, a w przypadku fałszu drugi argument.

```
let prawda x y = x;;
let fałsz x y = y;;
let jesli w k a = force (w k a);;
jesli prawda (lazy 1) (lazy 2);;
= 1
let i x y = x y fałsz;;
let lub x y = x prawda y;;
let nie x a b = x b a;;
jesli (lub fałsz prawda) (lazy 1) (lazy 2);;
= 1
```

26.4 Produkty kartezjańskie

Aby mówić o produkcie musimy mieć:

- konstruktor pary $\text{pr}: \alpha \rightarrow \beta \rightarrow \text{produkt}$,
- rzuty: $\text{p1}: \text{produkt} \rightarrow \alpha$ i $\text{p2}: \text{produkt} \rightarrow \beta$.

Produkt, to para argumentów czekających na funkcję.

```
let pr x y = function f ->f x y;;
let p1 p = p prawda;;
let p2 p = p fałsz;;
let p x = pr 1 "ala" x;;
p1 p;;
= 1
p2 p;;
= "ala"
```

Ze względu na system typów w Ocamlu, para p musi być zdefiniowana przy użyciu dodatkowego argumentu x (η -ekspansja).

26.5 Listy nieskończone

Listy nieskończone można reprezentować jako funkcje z `int`. Potrzebujemy:

- listę stałą,
- dołączenie elementu na początek listy,
- pierwszy element listy,
- ogon listy.

```
let stala x n = x;;
let sklej h t n = if n = 0 then h else t (n-1);;
let glowa l = l 0;;
let ogon l n = l (n + 1);;
```

26.6 Liczby naturalne Churcha

Pozostały nam jeszcze liczby naturalne. (Rozszerzenie liczb naturalnych do całkowitych pozostawiamy jako ćwiczenie dla ambitnych. :-) Utożsamiamy liczbę naturalną z liczbą iteracji zadanej funkcji.

$$n \equiv f \rightarrow f^n$$

```
let id x = x;;
let compose f g = function x ->f (g x);;
let zero f = id;;
let jeden f = f;;           = id
```

Dodawanie i mnożenie:

```
let inc n f = compose (n f) f;;
```

$$f^{n+1} = f^n \circ f$$

```
let plus m n f = compose (m f) (n f);;
```

$$f^{m+n} = f^m \circ f^n$$

```
let razy = compose;;
```

$$\begin{aligned} (\text{razy } m \ n) \ f &= (\text{compose } m \ n) \ f = \\ &= m \ (n \ f) = m \ (f^n) = \\ &= (f^n)^m = f^{m \cdot n} \end{aligned}$$

```
let potega m n = n m;;
```

$$\begin{aligned} (\text{potega } m \ n) \ f &= (n \ m) \ f = \\ &= \underbrace{(m \circ \dots \circ m)}_{n \text{ razy}} \ f = \\ &= \underbrace{m(m \dots (m \ f) \dots)}_{n \text{ razy}} = \\ &= \underbrace{m(m \dots (m \ f^m) \dots)}_{n-1 \text{ razy}} = \\ &= \underbrace{m(m \dots (m \ f^{m^2}) \dots)}_{n-2 \text{ razy}} = \\ &\quad \vdots \\ &= f^{m^n} \end{aligned}$$

Do celów testowych można używać:

```
let ile n = n (function x ->x + 1) 0;;
let compose f g = function x ->f (g x);;
let rec iterate n f =
  if n = 0 then id else compose (iterate (n-1) f) f;;
ile (iterate 1000);;
= 1000
```

```
let dwa f = inc jeden f;;
ile (razy dwa dwa);;
- : int = 4
```

```

let trzy f = plus dwa jeden f;;
let szesc f = razy dwa trzy f;;
let siedem f = inc szesc f;;
ile (razy szesc siedem);;
- : int = 42

```

```

let osiem f = potega dwa trzy f;;
ile (plus dwa (razy (plus dwa trzy) osiem));;
- : int = 42

```

Jak moglibyśmy odróżnić od siebie liczby naturalne Churcha? Podstawową operacją porównania jest tzw. test zera — sprawdzenie, czy dana liczba jest równa zero. Szukamy więc takiej funkcji f , że $f^0 = \text{id}$ jest czymś istotnie innym, niż f^i dla $i > 0$. Taką funkcją może być:

$$f(x) = \text{falsz}$$

```

let test_zera x = x (function _ ->falsz) prawda;;
jesli (test_zera zero) (lazy 1) (lazy 2);;
= 1

```

Jak zmniejszyć n o 1? Liczba n to taki obiekt, który przekształca zadaną funkcję f w funkcję f^n . Nie mamy jednak żadnego dostępu do tego jak n to robi. Problem ten można przedstawić sobie tak: mając dane n , f i x należy obliczyć $f^{n-1}(x)$.

Rozważmy funkcję:

$$g(a, b) = (b, f b)$$

oraz ciąg:

$$(g^i(x, x))_{i=0, \dots, n} = ((x, x), (x, f x), (f x, f^2 x), \dots, (f^{n-1} x, f^n x))$$

Wystarczy więc z ostatniej pary w ciągu wydobyć pierwszą współrzędną. Co można zapisać tak:

```

let dec n f x =
  let g (a,b) = (b,f b)
  in let (y,_) = n g (x,x)
      in y;;
val dec: (('a*'b->'b*'c)->'d*'d->'e*'f)->('b->'c)->'d->'e

```

lub tak, wykorzystując zaimplementowane przez nas produkty kartezjańskie:

```

let dec n f x = p1 (n (function p ->pr (p2 p) (f (p2 p))) (pr x x));;
val dec :
(((('a ->'b ->'b) ->'c) ->('c ->'d ->'e) ->'e) ->
(('f ->'f ->'g) ->'g) ->('h ->'i ->'h) ->'j) ->
('c ->'d) ->'f ->'j = <fun>
ile (dec dwa);;
= 1

```

Odejmowanie, to wielokrotne zmniejszanie o jeden:

```
let minus m n = (n dec) m;;
ile (minus dwa jeden);;
= 1
```

Za pomocą odejmowania i testu zera można zaimplementować porównanie:

```
let wieksze m n = nie (test_zera (minus m n));;
jesli (wieksze dwa jeden) (lazy 1) (lazy 2);;
= 1
```

26.7 Abstrakcja rekurencji

Procedura `fold_left` stanowi abstrakcję rekurencyjnego przetwarzania list. Czy istnieje kwintesencja wszelkiej rekurencji? A co to jest rekurencja? Jak przerobić definicję rekurencyjną na nierekurencyjną — poprzez wprowadzenie dodatkowego parametru funkcyjnego. Przykład: silnia. Z rekurencją:

```
let rec fac n =
  if n <= 1 then
    1
  else
    n * fac (n - 1);;
```

Bez rekurencji:

```
let factorial fac n =
  if n <= 1 then
    1
  else
    n * fac (n - 1);;
```

To jest procedura, która przetwarza procedurę `fac` liczącą poprawnie silnię dla liczb $\leq n$ na procedurę liczącą poprawnie silnię dla liczb $\leq n + 1$. Teraz trzeba jakoś chytrze podstawić funkcję wynikową jako argument. Inaczej mówiąc, szukamy funkcji, która jest punktem stałym `factorial`. Będzie ona poprawnie liczyć silnię dla wszystkich liczb.

Operator punktu stałego — pierwsze podejście.

```
let rec y f = f (y f);;
```

```
let f = y factorial ;;
= factorial (y factorial) = factorial (factorial (y factorial)) = ...
```

To się niestety zapętli, ze względu na to, że argument procedury `factorial` jest obliczany przed jej zastosowaniem. Aby uniknąć zapętlenia, musimy uleniwić ten argument i wyliczać go tylko na żądanie. Operator punktu stałego.

```

let factorial fac n =
  if n <= 1 then
    1
  else
    n * (force fac (n - 1));;

let rec y f = f (lazy (y f));;

let fac = y factorial ;;

```

$$\begin{aligned}
\text{fac } 3 &= (\text{y factorial}) 3 = \\
&= (\text{factorial}(\text{lazy}(\text{y factorial}))) 3 = \\
&= 3 * (\text{force}(\text{lazy}(\text{y factorial})) 2) = \\
&= 3 * (\text{y factorial } 2) = \\
&= 3 * (\text{factorial}(\text{lazy}(\text{y factorial})) 2) = \\
&= 3 * (2 * (\text{force}(\text{lazy}(\text{y factorial})) 1)) = \\
&= 3 * (2 * (\text{y factorial } 1)) = \\
&= 3 * (2 * (\text{factorial}(\text{lazy}(\text{y factorial})) 1)) = \\
&= 3 * (2 * 1) = 6
\end{aligned}$$

Każdą procedurę rekurencyjną możemy sprowadzić do zastosowania operatora punktu stałego i uleniwiania.

Oto kolejny przykład, rekurencyjna procedura obliczająca liczby Fibonacciego:

```
let fibonacci fib n =  
  if n <2 then  
    n  
  else  
    (force fib (n-1)) + (force fib (n-2));;  
  
let fib = y fibonacci;;
```

$$\begin{aligned} \text{fib } 3 &= y \text{ fibonacci } 3 = \\ &= \text{fibonacci}(\text{lazy}(y \text{ fibonacci})) 3 = \\ &= \text{force}(\text{lazy}(y \text{ fibonacci})) 2 + \text{force}(\text{lazy}(y \text{ fibonacci})) 1 = \\ &= y \text{ fibonacci } 2 + y \text{ fibonacci } 1 = \\ &= \text{fibonacci}(\text{lazy}(y \text{ fibonacci})) 2 + \text{fibonacci}(\text{lazy}(y \text{ fibonacci})) 1 = \\ &= \text{force}(\text{lazy}(y \text{ fibonacci})) 1 + \text{force}(\text{lazy}(y \text{ fibonacci})) 0 + 1 = \\ &= y \text{ fibonacci } 1 + y \text{ fibonacci } 0 + 1 = \\ &= \text{fibonacci}(\text{lazy}(y \text{ fibonacci})) 1 + \text{fibonacci}(\text{lazy}(y \text{ fibonacci})) 0 + 1 = \\ &= 1 + 0 + 1 = 2 \end{aligned}$$

Ćwiczenia

1. Zaimplementuj pary liczb całkowitych za pomocą operacji arytmetycznych i liczb całkowitych.
2. Dla dowolnych dwóch funkcji $f : X \rightarrow A$ i $g : X \rightarrow B$ istnieje dokładnie jedna taka funkcja $h : X \rightarrow A \times B$, że $\pi_1 \circ h = f$ i $\pi_2 \circ h = g$. Zdefiniuj wprost procedurę **prod**, która na podstawie funkcji f i g wyznacza funkcję h dla proceduralnej definicji produktów. (let prod f g x p = p (f x) (g x);;)
3. Zaimplementuj sumy rozłączne (koprodukty) za pomocą procedur. (Koprodukt zbiorów A i B to taki zbiór $A + B$ wraz z włożeniami $i_A : A \rightarrow A + B$, $i_B : B \rightarrow A + B$, że dla dowolnej pary funkcji $f : A \rightarrow X$ i $g : B \rightarrow X$ istnieje dokładnie jedna taka funkcja $h : A + B \rightarrow X$, że $h \circ i_A = f$ i $h \circ i_B = g$. Potraktuj tę definicję dosłownie.) Należy zaimplementować włożenie A i B w $A + B$ oraz procedurę, która na podstawie funkcji f i g wyznaczy funkcję h .
Na potrzeby tego zadania możesz przyjąć, że A i B to ten sam typ.
4. Potęgowanie funkcji w czasie logarytmicznym. Co tak na prawdę jest obliczane szybciej: funkcja wynikowa, czy jej wartość?
5. Jak można rozszerzyć liczby naturalne Churcha do liczb całkowitych?
6. Jak za pomocą operatora punktu stałego można wyznaczać procedury wzajemnie rekurencyjne?

Wykład 27. Strumienie

[[[Scałic z materiałami z Ważniaka.]]]

27.1 Czas rzeczywisty, a czas symulowany

Programowanie imperatywne i analogia obiektowa. Czas w modelowanym systemie jest modelowany przez czas w modelu systemu.

Programowanie strumieniowe — nie ma takiej analogii z czasem. Staramy się w modelu oddać strukturę zależności, a niekoniecznie kolejność zdarzeń.

Strumień to ciąg wartości — cała, być może nieskończona, historia obiektu. Zamiast mówić o akcjach i zmianach stanu obiektów w danych chwilach patrzymy „jednym rzutem oka” na całą historię obiektu w czasoprzestrzeni. Staramy się oddać zależności między liniami świata jednych obiektów i innych.

27.2 Implementacja strumieni

Formalnie strumień to ciąg — być może nieskończony. Pomysł polega jednak na tym, żeby był on „leniwy”, tzn. obliczane były tylko potrzebne wartości. Tak więc strumień to para: wartość i odroczonego strumień pozostałych wartości.

```
type 'a stream = Nil | Cons of 'a * 'a stream Lazy.t;;
```

```
let empty s = s = Nil;;
```

```
let head s =  
  match s with  
  Nil ->failwith "Empty" |  
  Cons (h,_) ->h;;
```

```
let tail s =  
  match s with  
  Nil ->failwith "Empty" |  
  Cons (_,t) ->force t;;
```

```
EXTEND ... Makro Cons
```

Mając do dyspozycji takie konstruktory i selektory możemy zdefiniować kilka pomocniczych operacji na strumieniach:

```
let rec const_stream c =  
  Cons (c,const_stream c);;
```

```
let ones = const_stream 1;;
```

```
let rec filter_stream p s =  
  if empty s then  
    Nil  
  else  
    let h = head s
```

```

    and t = tail s
  in
    if p h then
      Cons (h,filter_stream p t)
    else
      filter_stream p t;;

let rec stream_ref s n =
  if n = 0 then
    head s
  else
    stream_ref (tail s) (n - 1);;

let rec stream_map f s =
  if empty s then
    Nil
  else
    Cons (f (head s),stream_map f (tail s));;

let rec stream_fold f a s =
  if empty s then
    Cons (a,Nil)
  else
    Cons (a,stream_fold f (f a (head s)) (tail s));;

let rec for_each p s =
  if not (empty s) then begin
    p (head s);
    for_each p (tail s)
  end;;

let print_int_stream s =
  let p x =
    begin
      print_int x;
      print_string "\ n";
      flush stdout
    end
  in
    for_each p s;;

```

27.3 Przykłady strumieni nieskończonych

Spróbujmy zdefiniować strumień liczb naturalnych:

```

let rec integers_from x =
  Cons(x,integers_from (x+1));;

let nats = integers_from 0;;

```

Podobnie możemy zdefiniować strumień liczb Fibonacciego:

```
let fibs =
  let rec fibo a b =
    Cons (a,fibo b (a+b))
  in fibo 0 1;;
```

Jak widać, jeżeli jesteśmy w stanie skonstruować iteracyjną procedurę wyliczającą kolejne elementy strumienia, to tym samym jesteśmy w stanie zdefiniować strumień tych wartości. Procedura taka stanowi ukrytą w strumieniu „maszynę”, która na żądanie podaje kolejne elementy strumienia. Takie definicje strumieni nazywamy *niewiukłanymi* (w odróżnieniu od *wiukłanych*, które poznamy dalej). W przypadku definicji niewiukłanych bytem rekurencyjnym jest procedura, a nie strumień. Strumień jest wtórny wobec procedury rekurencyjnej.

27.4 Strumień liczb pierwszych — sito Eratostenesa

Spróbujmy skonstruować nieskończony strumień liczb pierwszych, generowanych metodą sita Eratostenesa. Sposób konstrukcji takiego strumienia możemy przedstawić w postaci schematu przypominającego schematy blokowe układów przetwarzających sygnały. Strumienie są zaznaczone liniami ciągłymi, a pojedyncze wartości przerywanymi.

Schemat ten możemy zapisać w postaci następującego programu:

```
let divisible x y = x mod y = 0;;

let sito p s =
  filter_stream (function x ->not (divisible x p)) s;;

let rec sieve s =
  Cons (head s,sieve (sito (head s) (tail s)));;

let primes =
  sieve (integers_from 2);;
```

27.5 Definicje uwikłane

Definiując nieskończone strumienie nie musimy tego zawsze robić poprzez podanie odpowiedniej procedury rekurencyjnej (*integers-from*, *fibgen*, *sieve*). Możemy użyć elementów strumienia do zdefiniowania jego samego.

```
let rec ones =
  Cons (1,ones);;
```

Do budowy bardziej skomplikowanych strumieni potrzebujemy dodatkowych operacji budujących strumienie. Na przykład, do utworzenia strumienia liczb naturalnych czy liczb Fibonacciego potrzebujemy dodawania strumieni.

```
let rec add_int_streams s1 s2 =
  if empty s1 || empty s2 then
    Nil
  else
    Cons (head s1 + head s2, add_int_streams (tail s1) (tail s2));;

let rec nats =
  Cons (1, add_int_streams ones nats);;
```

Zamiast dodawania strumienia jedynek, możemy użyć operacji zwiększania o jeden:

```
let succ x = x + 1;;
let rec nats =
  Cons(0, stream_map succ nats);;
```

A oto uwikłana definicja liczb Fibonacciego:

```
let rec fibs =
  Cons (0, Cons (1, add_int_streams fibs (tail fibs)));;
```

Możemy też w uwikłany sposób zdefiniować strumień liczb pierwszych. Użyjemy do tego predykatu sprawdzającego, czy liczba jest pierwsza:

```
let rec primes =
  Cons (2, filter_stream prime (integers_from 3))
```

Natomiast predykat `prime?` zdefiniujemy używając ... strumienia liczb pierwszych:

```
and prime n =
  let rec iter ps =
    if square (head ps) > n then
      true
    else if divisible n (head ps) then false
    else iter (tail ps)
  in
    iter primes;;
```

Całość działa poprawnie, ponieważ strumień jest leniwą strukturą danych. Pierwszy element, 2, jest dany *explicit*e. Natomiast sprawdzając kolejne liczby, czy są pierwsze, zawsze mamy już obliczone potrzebne liczby pierwsze. Konkretnie, największa obliczona liczba pierwsza zawsze jest większa od pierwiastka z kolejnej liczby pierwszej, $2^2 = 4 > 3$, $3^2 = 9 > 5$, ...

27.6 Iteracje jako strumienie

W przypadku definicji niewykłanych używaliśmy procedur iteracyjnych do zdefiniowania strumieni. Możemy ten mechanizm odwrócić i użyć strumieni do opisanego procesów iteracyjnych — kolejne elementy strumienia mogą reprezentować kolejne kroki iteracji. Oto przykład, strumień kolejnych przybliżeń pierwiastka, metodą Newtona:

```
let sqrt_improve g x = (g +. x /. g) /. 2.0;;

let sqrt_stream x =
  let rec guesses =
    Cons (1.0,
         stream_map (function guess ->sqrt_improve guess x) guesses)
  in guesses;;
```

Spróbujmy przybliżyć liczbę π . Użyjemy do tego celu szeregu zbieżnego do $\frac{\pi}{4}$:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$$

```
let scale_stream c s =
  stream_map (function x ->x *. c) s;;

let pi_summands =
  let
    succ x =
      if x >0.0 then
        -.x -. 2.0
      else
        -.x +. 2.0
  in
    let rec s =
      Cons (1.0,stream_map succ s)
    in
      stream_map (fun x ->1.0 /.x) s;;

let partial_sums s =
  let rec ps =
    Cons (head s,add_float_streams (tail s) ps)
  in ps;;

let pi_stream = scale_stream 4.0 (partial_sums pi_summands);;
```

Taki strumień jest co prawda zbieżny, ale bardzo wolno. Po zsumowaniu pierwszych 1000 elementów ustalone są dopiero trzy pierwsze cyfry: 3.14. Jest to dosyć oczywiste, jeżeli zauważymy, że suma pierwszych n elementów jest obarczona błędem rzędu $\frac{1}{n}$.

Można „przyspieszyć” zbieżność tego szeregu stosując „akcelerator Eulera”. Jeśli S_n jest sumą pierwszych n wyrazów szeregu, to szereg zakcelerowany ma postać.

$$S_{n+1} - \frac{(S_{n+1} - S_n)^2}{S_{n-1} - 2S_n + S_{n+1}}$$

Działa on dobrze dla ciągów o malejących modułach błędów przybliżenia.

Dlaczego on działa? Przedstawmy kolejne sumy częściowe szeregu w postaci granica plus błąd: $S_n = x + e_n$, $|e_n| > |e_{n+1}| > 0$. Wówczas elementy przyspieszonego szeregu mają postać:

$$\begin{aligned} x + e_{n+1} - \frac{(x + e_{n+1} - x - e_n)^2}{x + e_{n-1} - 2x - 2e_n + x + e_{n+1}} &= \\ &= x + e_{n+1} - \frac{(e_{n+1} - e_n)^2}{e_{n-1} - 2e_n + e_{n+1}} = \\ &= x + \frac{e_{n+1}e_{n-1} - 2e_{n+1}e_n + e_{n+1}^2 - e_n^2 + 2e_n e_{n+1} - e_n^2}{e_{n-1} - 2e_n + e_{n+1}} = \\ &= x + \frac{e_{n+1}e_{n-1} - e_n^2}{e_{n-1} - 2e_n + e_{n+1}} \end{aligned}$$

Jeżeli np. $|e_n| = e \cdot c^n$ i znaki e_n są takie same lub naprzemienne, to $e_{n+1}e_{n-1} - e_n^2 = 0$, czyli ciąg natychmiast osiąga granicę.

[[... sprawdzić, kiedy można strumień wiele razy przyspieszać ...]]

Oto implementacja akceleratora Eulera:

```
let rec euler_transform s =
  let s0 = stream_ref s 0
  and s1 = stream_ref s 1
  and s2 = stream_ref s 2
  in Cons
    (s2 -. square (s2 -. s1) /. (s0 -. 2.0 *. s1 +. s2),
     euler_transform (tail s));;
```

Taki strumień jest już zbieżny w rozsądnym czasie. Przeliczenie 1000 elementów przyspieszonego szeregu daje przybliżenie 3.141592653. Jeszcze lepsze wyniki daje wielokrotne przyspieszanie. Skonstruujmy strumień kolejnych przyspieszeń strumienia sum częściowych i wybierzemy z niego strumień pierwszych elementów:

```
let rec pi_table =
  Cons (pi_stream, stream_map euler_transform pi_table);;

let pi_stream = stream_map head pi_table;;

let pi = stream_ref pi_stream 8;;
```

Taki strumień już w ósmym elemencie osiąga granice precyzji arytmetycznej.

27.7 Strumienie par — liczby Ramanujana

Przedstawimy teraz jako przykład operowania strumieniami, konstrukcję strumienia liczb Ramanujana. Liczby Ramanujana to takie liczby naturalne, które można przedstawić jako sumy sześciątów dwóch liczb naturalnych na dwa różne sposoby. Pierwsze z tych liczb, to: 1729, 4104, 13832, 20683. Niech waga pary liczb naturalnych to będzie suma sześciątów jej składowych. Pomysł konstrukcji strumienia liczb Ramanujana polega na skonstruowaniu strumienia

(nieuporządkowanych) par liczb naturalnych uporządkowanego wg niemalejących wag, wychwyceniu par o powtarzających się wagach i wypisaniu tych wag.

Następująca procedura łączy dwa strumienie uporządkowane wg niemalejących wag elementów w jeden uporządkowany strumień. Waga jest określona w postaci funkcji w

```
let rec merge_weighted w s1 s2 =
  if empty s1 then s2 else
  if empty s2 then s1 else
  let h1 = head s1
  and h2 = head s2
  in
    if w h1 <w h2 then
      Cons (h1,merge_weighted w (tail s1) s2)
    else if w h1 >w h2 then
      Cons (h2,merge_weighted w s1 (tail s2))
    else
      Cons (h1,Cons (h2,merge_weighted w (tail s1) (tail s2))));;
```

Następujące definicje określają wagi par oraz funkcję tworzącą strumień par uporządkowanych wg wag.

```
let cube x = x * x * x;;
```

```
let weight (x,y) = cube x + cube y;;
```

```
let rec pairs s =
  Cons ((head s,head s),
    merge_weighted
      weight
      (stream_map (function x ->(head s,x)) (tail s))
      (pairs (tail s))));;
```

Następująca procedura pozostawia jedynie reprezentantów sekwencji elementów o powtarzających się wagach.

```
let rec non_uniq w s =
  let rec skip we s =
    if empty s then Nil
    else if we = w (head s) then skip we (tail s)
    else s
  in
    if empty s then Nil
    else if empty (tail s) then Nil
    else
      let h1 = head s
      and h2 = head (tail s)
      in
        if w h1 = w h2 then
          Cons (w h1,non_uniq w (skip (w h1) s))
        else
          non_uniq w (tail s));;
```

Strumień liczb Ramanujana możemy zdefiniować następująco:

```
let ramanujan =  
  non_uniq_weight (pairs nats);;
```

27.8 Co jeszcze ...

Strumienie jako reprezentacja danych, np. szeregów potęgowych. Operacje na funkcjach reprezentowanych w taki sposób.

Ćwiczenia

Zdefiniuj strumienie i narysuj schematy odpowiadające ich definicjom. (Tam, gdzie się da, w sposób uwikłany.)

1. Strumień silni.
2. Strumień postaci: jedna jedynka, dwie dwójki, trzy trójki itd. (Do rozwiązania uwikłanego potrzebne jest scalanie strumieni monotonicznych.)
3. Napisz procedurę `codrugi`: `'a stream -> 'a stream`, która przekształca strumień postaci $(s_1; s_2; s_3; s_4; \dots)$ w strumień postaci $(s_1; s_3; s_5; \dots)$. Powinna ona działać zarówno dla strumieni skończonych, jak i nieskończonych.
4. Przeplot elementów dwóch strumieni (można sprytnie, zamieniając w wywołaniu rekurencyjnym miejscami argumenty).
5. Strumień wszystkich par (uporzędkowanych) elementów z dwóch danych strumieni (w dowolnej kolejności).
6. Strumień liczb całkowitych, które w rozkładzie na liczby pierwsze mają tylko 2, 3 i 5 [R.Hamming].
7. Dany jest nieskończony strumień liczb $s = (s_1, s_2, s_3, \dots)$. Jego strumień różnicowy, to strumień postaci: $s' = (s_2 - s_1, s_3 - s_2, s_4 - s_3, \dots)$. Strumień różnicowy drugiego rzędu, to strumień różnicowy strumienia różnicowego. Ogólniej, strumień różnicowy n -tego rzędu polega na n -krotnym wzięciu strumienia różnicowego, zaczynając od s . Zdefiniuj, w sposób uwikłany, strumień złożony z pierwszych elementów strumieni różnicowych kolejnych rzędów $(s_1, s_2 - s_1, (s_3 - s_2) - (s_2 - s_1), \dots)$. Narysuj diagram ilustrujący rozwiązanie.
8. [IOI 2005] Uśrednienie strumienia $s = (s_1, s_2, \dots)$, to strumień (o jeden element krótszy) postaci: $(\frac{s_1+s_2}{2}, \frac{s_2+s_3}{2}, \dots)$.
Dla danego strumienia liczb naturalnych s oblicz taki strumień (x_1, x_2, \dots) , że x_i jest liczbą takich niemalejących strumieni liczb całkowitych $(y_1, y_2, \dots, y_{i+1})$, których uśrednieniem jest (s_1, s_2, \dots, s_i) .
9. Dany jest nieskończony strumień nieskończonych strumieni s . Zdefiniuj jego „przekątną” p .
10. *Słowa Fibonacciego* to ciąg napisów zdefiniowany następująco: $F_1 = b, F_2 = a$, dla $i > 2$ słowo F_i powstaje ze sklejenia słów F_{i-1} i F_{i-2} . Oto początkowy fragment tego ciągu:

i	F_i
1	b
2	a
3	ab
4	aba
5	$abaab$
6	$abaababa$

Zauważ, że każde słowo Fibonacciego F_i (dla $i > 1$) jest prefiksem kolejnego słowa Fibonacciego F_{i+1} . Tak więc, wszystkie te słowa są prefiksami pewnego nieskończonego ciągu znaków a i b . Zdefiniuj strumień tych znaków.

11. Szereg potęgowy $a_0 + a_1x + a_2x^2 + \dots$ możemy reprezentować jako strumień jego kolejnych współczynników; przy takiej implementacji szeregów potęgowych zaimplementuj:

- pochodną,
- całkę,
- interpolacje wybranych funkcji (np.: e^x , $\ln x$, $\cos x$, $\sin x$),
- mnożenie szeregów potęgowych,
- niech X będzie szeregiem potęgowym o pierwszym współczynniku równym 1; oblicz odwrotność X , tzn. taki szereg S , że $X \cdot S = 1$; niech $X = 1 + x \cdot X'$, wówczas:

$$(1 + x \cdot X') \cdot S = 1$$

$$S + x \cdot X' \cdot S = 1$$

$$S = 1 - x \cdot X' \cdot S$$

- korzystając z wyników poprzedniego zadania zaimplementuj dzielenie szeregów potęgowych.

12. Na podstawie wzoru: $e = \sum_{i=0}^{\infty} \frac{1}{i!}$ skonstruować strumień kolejnych cyfr liczby e .

Wykład 28. Programowanie obiektowe

28.1 Ideologia programowania obiektowego

- Rozszerzenie paradygmatu programowania imperatywnego. Obiektom z modelowanego problemu odpowiadają obiekty obliczeniowe.
- Wyróżniamy klasy obiektów z modelowanego problemu. Między obiektami (lub ich klasami) zachodzą różnego rodzaju zależności. Typowe rodzaje zależności, to:
 - relacja zawierania,
 - relacja znajomości,
 - relacja dziedziczenia.
- Relacje zawierania i znajomości to relacje między pojedynczymi obiektami. Są one realizowane w ten sposób, że jeden obiekt obliczeniowy posiada dowiązanie do drugiego obiektu.
- Relacja dziedziczenia, to relacja między klasami obiektów.
- Metodologia projektowania i programowania obiektowego (w skrócie):
 - Wyróżnienie (klas) obiektów występujących w opisie problemu.
 - Projekt hierarchii klas i zależności między obiektami tworzącymi model danych.
 - Analiza przypadków użycia i typowych interakcji z systemem.
 - Dodanie obiektów sterujących i realizujących interfejs.
 - Analiza interakcji (komunikacji) między obiektami i metod potrzebnych do zrealizowania systemu.
 - Implementacja.

Przykład: Dokonać analizy hierarchii klas i zależności między obiektami dla gry będącej tematem programu zaliczeniowego.

28.2 Programowanie obiektowe

28.2.1 Klasy i obiekty

Obiekty (tej samej klasy) możemy traktować jak rekordy zawierające:

- pola z atrybutami,
- metody dostępu do obiektu.

Definicja klasy ma postać:

```
class punkt =  
  object  
    val mutable x = 0
```

```

val mutable y = 0
method polozenie = (x,y)
method przesun (xv,yv) =
  begin
    x <- x + xv;
    y <- y + yv
  end
end;;

```

Z obiektu korzystamy w następujący sposób:

```

let p = new punkt;;
p#przesun (1,0);;
p#przesun (3,2);;
p#polozenie;;

```

Treść definicji klasy jest wykonywana za każdym razem, gdy tworzony jest obiekt. Czasami chcemy, aby inicjacja obiektu (konstruktor) była sparametryzowana. Możemy to uzyskać parametryzując klasę. Przy tym parametry inicjacji są widoczne przez cały czas życia obiektu.

```

class punkt (x0,y0) =
  object
    val mutable x = x0
    val mutable y = y0
    method polozenie = (x,y)
    method przesun (xv,yv) =
      begin
        x <- x + xv;
        y <- y + yv
      end
    method na_poczatek =
      begin
        x <- x0;
        y <- y0
      end
  end
end;;

```

```

let p = new punkt (8,5);;
p#przesun (-4,-3);;
p#polozenie;;
p#na_poczatek;;
p#polozenie;;

```

28.2.2 Dziedziczenie

Istotnym elementem obiektowego paradygmatu programowania jest dziedziczenie. Dziedziczenie polega na tym, że definiując nową klasę możemy określić, że stanowi ona wzbogacenie istniejącej już klasy — dziedziczy jej cechy i metody. Oznacza to, że nowo definiowana klasa posiada wszystkie atrybuty i metody klasy, z której dziedziczy.

```

class kolorowy_punkt p c =
  object
    val color : int = c
    inherit punkt p
    method kolor = color
  end;;

let p = new kolorowy_punkt (3,1) 42;;
p#przesun (1,1);;
p#polozenie;;
p#kolor;;

```

Typ klasy dziedziczącej i tej, z której się dziedziczy są różne. W pewnym jednak sensie możemy traktować typ klasy dziedziczącej jak podtyp klasy, z której dziedziczy. Dlatego też mówimy o nadklasach i podklasach.

Możliwe jest również wielodziedziczenie — po prostu w treści definicji klasy może się pojawić wiele klauzul `inherit`.

Z pojęciem dziedziczenia związane jest pojęcie rzutowania typów. Typ podklasy można rzutować na typ nadklasy. Operator rzutowania typów ma postać:

```
(p : kolorowy_punkt ->punkt)
```

Jeżeli typ rzutowanego obiektu jest znany, to można go pominąć:

```
(p ->punkt)
```

28.2.3 Metody wirtualne

Metody wirtualne, to sposób tworzenia abstrakcyjnych nadklas. Metoda wirtualna ma określony typ, ale nie ma implementacji. Implementacja taka musi być zdefiniowana w podklasach. Dopóki wszystkie metody wirtualne nie zostaną zinstancjonowane, nie można tworzyć obiektów danej klasy.

```

class virtual punkt =
  object
    val mutable x = 0
    val mutable y = 0
    method polozenie = (x,y)
    method virtual przesun : int * int ->unit
  end;;

class p =
  object
    inherit punkt
    method przesun (xm,ym) =
      begin
        x <- x + xm;
        y <- y + ym
      end
  end;;

```

Ciekawe zadania:

- pręt i łuk,
- stacje benzynowe z różnymi cenami.

Wykład 29. Co jeszcze?

- Uruchamianie, testowanie i asercje w programach. Profilowanie programów.
- Pliki
- atrapy i strażnicy
- obiegi drzew,
- konwersja wyrażeń z postaci infiksowej na prefiksową i postfiksową (ONP),
- problem plecakowy,
- optymalne mnożenie wielu macierzy.

Literatura

- [Apt81] Krzysztof R. Apt. Ten years of Hoare's logic: A survey — part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.
- [BA05] Mordechai Ben-Ari. *Logika matematyczna w informatyce*. WNT, 2005.
- [Ben08] Jon Bentley. *Perelki Oprogramowania*. WNT, Warszawa, 3 edition, 2008.
- [Bra83] J. M. Brady. *Informatyka teoretyczna w ujęciu programistycznym*. WNT, 1983.
- [BS95] Holger Bickel and Werner Struckmann. The Hoare logic of data types. Technical Report 95-04, Technische Universität Braunschweig, Deutschland, 1995.
- [CMP] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. Developing applications with objective caml. <http://caml.inria.fr/pub/docs/oreilly-book/>.
- [CO81] Robert Cartwright and Derek Oppen. The logic of aliasing. *Acta Informatica*, 15:365–384, 1981.
- [Dah92] Ole-Johan Dahl. *Verifiable Programming*. Prentice Hall, 1992.
- [Dij85] Edsger W. Dijkstra. *Umiejętność programowania*. Wydawnictwa Naukowo-Techniczne, 1985.
- [HA02] G. J. Sussman H. Abelson. *Struktura i interpretacja programów komputerowych*. WNT, 2002.
- [Her65] H. Hermes. *Enumerability, Decidability and Computability*. Springer, 1965.
- [Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.
- [Kub] Marcin Kubica. Programowanie funkcyjne. http://wazniak.mimuw.edu.pl/index.php?title=Programowanie_funkcyjne.
- [Kub00] Marcin Kubica. *Formalna specyfikacja wskaźnikowych struktur danych*. PhD thesis, Wydział Matematyki, Informatyki i Mechaniki, Uniwersytet Warszawski, 2000.
- [Kub03] Marcin Kubica. Temporal approach to specification of pointer data-structures. In Mauro Pezzè, editor, *Fundamental Approaches to Software Engineering, FASE'2003*, number 2621 in LNCS, pages 231–245. Springer-Verlag, 2003.
- [Ler] Xavier Leroy. The objective caml system. <http://caml.inria.fr/pub/docs/manual-ocaml/index.html>.
- [MS87] Grażyna Mirkowska and Andrzej Salwicki. *Algorithmic Logic*. Państwowe Wydawnictwo Naukowe, 1987.
- [MS92] Grażyna Mirkowska and Andrzej Salwicki. *Logika algorytmiczna dla programistów*. Wydawnictwa Naukowo-Techniczne, 1992.

[[Dodać CLR i ew. inne niewstawione skróty.]]