Wstęp do programowania w wydaniu funkcyjnym

Marcin Kubica

2002/2003

1 Wstęp

1.1 „Wstęp do programowania” 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 bezpośrednie 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 iście 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 *Scheme*. 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 Musa Al-Chuwarizmi ok. 780–850 r. n.e. matematyk, geograf i astronom działający w Bagdadzie na dworze Haruna Al-Raszida. Autor:

- *Tablic* (astronomia, kalendarz, funkcje trygonometryczne),
- *Arytmetyki* (hinduski zapis pozycyjny, metody rachunkowe),
- *Algebry* (równania liniowe i kwadratowe),
- *Kalendarza żydowskiego*, *Kronik*, *Geografii* i *Astrologii*.

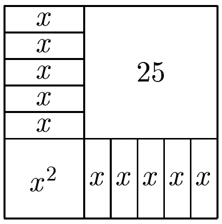
Jego prace przyczyniły się do rozpowszechnienia cyfr arabskich i zapisu pozycyjnego. Od jego nazwiska pochodzi słowo **algorytm** (Alchwarizmi, Algorithmi, Algorithmus).

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

(Przygotować metody 2–5.)

Metodę 1 ilustruje następujący rysunek:



Pole dużego kwadratu wynosi $39 + 25 = 64 = 8 \times 8$. Stąd $x + 5 = 8$, czyli $x = 3$. Dla pozostałych przypadków mamy podobne metody.

1.4 Inne języki programowania

Programowanie było znane od dawna, choć nie wiedziano jeszcze, że jest to programowanie:

- Przepisy kulinarne:
  - przepis — program,
  - gotowanie — wykonanie,
  - potrawa — efekt.
- Zapis nutowy:
  - nuty — program,
  - taśma dziurkowana — program dla katarynki lub pianoli,
  - muzyka — efekt działania katarynki, pianoli, czy orkiestry,
  - przykład.
- Systemy składu tekstu (np. LaTeX):
  - plik tekstowy zawierający tekst i opis jak powinien on być złożony — program,
  - skład tekstu — wykonanie,
  - wydruk — efekt działania,
  - przykład.
- Opisy skalowalnych czcionek (np. MetaFont):
  - opis czcionek — program,
  - generowanie wyglądu czcionek przy zadanej rozdzielczości i dla zadanego urządzenia — wykonanie,
  - wygląd czcionek — efekt.

1.5 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 $\langle R, +, -, *, /, \surd, <, = \rangle$. 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.

1.6 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 dziedzina algorytmiczna jak i nasz cel są pojęciami matematycznymi. Zaklęcia nasze opisują, jak z prostszych pojęć matematycznych konstruować bardziej skomplikowane, aż uda nam się skonstruować zaklęcie opisujące cel. 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.

1.7 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 eksperyment — roczny.

Porównanie:

Imperatywne	Funkcyjne
Pascal (FPC)	Lisp (Scheme)
DOS	Unix

Jeśli nie jesteś pewien swoich sił, lepiej wybierz potok imperatywny. Jeśli nie znasz programowania, ale chcesz wziąć udział w eksperymencie, możesz spróbować programowania funkcyjnego. Jeśli znasz Pascala, to programowanie funkcyjne może być ciekawsze.

1.8 Literatura

H. Abelson, G. J. Sussman, *Struktura i interpretacja programów komputerowych*, WNT 2002.

1.9 Kwestie organizacyjne

- Zasady zaliczenia — 2 kolokwia.
- Po wyborze potoku nie będzie możliwości przeniesienia się.
- Wybór chętnych do grup funkcyjnych na wolne miejsca.

Laboratorium

Zapoznanie się ze środowiskiem: logowanie się, podstawowe polecenia Unixa, edytor emacs, joe, mc.

Ćwiczenia

- Wyznacz przecięcie dwóch prostokątów o bokach równoległych do osi współrzędnych i całkowitych współrzędnych wierzchołków. Nacisk na specyfikację — warunek wstępny na dane. Rozwiązanie w postaci funkcji matematycznych.

  Reprezentacja Punkt = Z^2, Prostokąt = Z^4. O interpretacji i reprezentacji, funkcja interpretacji/abstrakcji. Podział na podproblemy: rzutowanie, przecięcie odcinków na osi, budowa prostokąta na podstawie jego rzutów.

- Napisz funkcję określającą, czy dwa odcinki o końcach o współrzędnych całkowitych przecinają się. Dostępna jest tylko dziedzina algorytmiczna liczb całkowitych i wartości logicznych. Jak sprawdzić, czy dwa odcinki się przecinają. Po której stronie prostej leży punkt — iloczyn wektorowy. Uwaga na odcinki zdegenerowane.

  Reprezentacja: wektora swobodnego, odcinka (wektora zaczepionego) na płaszczyźnie. Podział na podproblemy: czy odcinek ma końce po dwóch stronach prostej wyznaczonej przez drugi odcinek (≤ 0), iloczyn wektorowy, prostokąt opisujący odcinek, niepusty prostokąt. W przypadkach brzegowych decyduje kryterium przecinania się prostokątów opisujących odcinki.

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.inn.: $+$, $-$, $*$, $/$, `mod`, 0, 1, 2, ..., `true`, `false`, `or`, `and`, `not`, $=$, $\leq$, $\geq$, $<$, $>$, $<>$. Powtórka: rozróżnienie symboli od ich interpretacji.

2.1 Wyrażenia

Najprostsza konstrukcja programistyczna — wyrażenia. Konstruując wyrażenia złożone posługujemy się tzw. notacją polską wraz z nawiasami wyznaczającymi liczbę argumentów, tzn. wyrażenie ma postać: ($operator\ argument_1\ \ldots\ argument_n$). Niektóre operatory mogą działać dla różnej liczby argumentów (np. +), stąd nawiasy są potrzebne. Kilka przykładów wyrażeń całkowitych (wszystkie poniższe wyrażenia są równe 486):

```
486
(+ 137 349)
(* 18 27)
(-1000 514)
(/ 5832 12)
(-(* 1 2 3 4 5 6)
   (/
     (-(* 7 8 9) (* 3 12))
     (-(* 3 4) (* 2 5))
   )
)
```

Jeden taki zestaw operator-argumenty ujęty w nawiasy będziemy nazywać *kombinacją*. Tak więc wyrażenie może mieć postać:

- liczby,
- nazwy stałej, lub
- kombinacji, czyli listy podwyrażeń, z których pierwsze jest traktowane jak operator, a kolejne jak argumenty.

Zauważmy, że operator i argumenty są tu traktowane na równi. Takie symbole jak `+` czy `*` to po prostu stałe, których wartościami są procedury wbudowane w język programowania, a `123` czy `486` to stałe liczbowe.

2.1.1 BNF i składnia wyrażeń

Gramatyka bezkontekstowa jako sposób opisu związku między zapisem, a drzewem rozbioru gramatycznego. (Nie trzeba być tu super formalnym. Zamiast definicji gramatyki, opis notacji.) BNF:

- $\langle$`konstrukcja`$\rangle$,
- ::=,
- <u>`słowo kluczowe`</u>,
- $\dots|\dots$,
- $[\dots]$,
- $\{\dots\}^*$,
- $\{\dots\}^+$,
- $\{\dots\}$.

Tego formalizmu będziemy używać do opisu składni. Przykład: składnia wyrażeń arytmetycznych.

$$
\begin{array}{lcl}
\langle\texttt{wyrażenie}\rangle & ::= & \langle\texttt{stała}\rangle|\langle\texttt{kombinacja}\rangle. \\
\langle\texttt{kombinacja}\rangle & ::= & \underline{(}\langle\texttt{wyrażenie}\rangle^{+}\underline{)}. \\
\langle\texttt{stała}\rangle & ::= & \langle\texttt{identyfikator}\rangle|\langle\texttt{liczba}\rangle. \\
\langle\texttt{liczba}\rangle & ::= & 0|1|2|\dots\ .
\end{array}
$$

Jak poprzedni przykład się wyprowadza w powyższej gramatyce.

2.1.2 Obliczanie wartości wyrażeń

Efektem „działania" wyrażenia jest jego wartość. Wyrażenie możemy sobie przedstawić jako drzewo — w liściach mamy stałe, a węzły wewnętrzne to kombinacje. Nawiasy wyznaczają kształt tego drzewa. Zilustrować na przykładzie powyższego wyrażenia. Sposób wyliczania wyrażenia jako dekorowanie drzewa wartościami. Wartość w korzeniu to wynik.

Możemy sformułować następujący model obliczeń. Mamy dane pewne *środowisko*, które nazwom przypisuje pewne stałe (znaczenie tych nazw). Algorytm wyliczenia wartości wyrażenia wygląda następująco:

- wartością liczby jest ona sama,
- wartość stałej jest pobierana ze środowiska,
- w celu obliczenia wartości kombinacji oblicz wartości wszystkich podwyrażeń tworzących kombinację, a następnie zastosuj wartość pierwszego podwyrażenia (powinna to być procedura) do wartości pozostałych podwyrażeń (jako jej argumentów).

2.2 Definicje stałych

Definiowanie stałych:

$$\langle\texttt{definicja}\rangle ::= \underline{\texttt{(define}}\ \langle\texttt{identyfikator}\rangle\ \langle\texttt{wyrażenie}\rangle\underline{\texttt{)}}.$$

Proste przykłady:

```
(define a 6)
(define b (+ a 1))
(* a b)
42
```

Definicje są traktowane inaczej niż wyrażenia. Powodują one zmianę środowiska. Dlatego też `define` jest nazywane *formą specjalną*. „Obliczenie” `define` wykonywane jest następująco:

- oblicz wartość wyrażenia tworzącego definicję,
- wstaw do środowiska identyfikator i przyporządkuj mu wartość obliczonego wyrażenia.

Jeżeli symbol jest już w środowisku, to można *przysłonić* go i nadać symbolowi nowe znaczenie. Uwaga: jest to uproszczony model, który w miarę potrzeb będziemy zmieniać.

```
(define a 2)
(define b (* 2 a)) = 4
(define a (* a b)) = 8
```

Eksplozja wyrażeń: jeżeli mamy ciąg definicji, kolejne korzystają z poprzednich, to możemy je rozwinąć. Rozwinięcie takie może mieć jednak wykładniczy rozmiar. Przykład.

2.3 Definicje procedur

Jak dotąd poznaliśmy trzy podstawowe konstrukcje językowe:

- wartości zdefiniowanej w środowisku i liczby (elementy dziedziny algorytmicznej),
- kombinacje i możliwość ich zagnieżdżania (sposób konstrukcji),
- definicje stałych (sposób abstrakcji).

Podejście funkcyjne można ująć w następujący sposób:

- procedury to też wartości, tyle, że z odpowiednich zbiorów funkcji częściowych (np. $*$ jest wartością ze zbioru funkcji `int` $*$ `int` $\to$ `int` (no prawie ;)), procedury są równie dobrymi wartościami jak inne wartości.

Połączenie tych dwóch zasad prowadzi do następujących wniosków:

- Możemy definiować wartości, które są procedurami.

  $$\langle\texttt{definicja}\rangle ::= \underline{\texttt{(define}}\ \underline{\texttt{(}}\langle\texttt{identyfikator}\rangle^{+}\underline{\texttt{)}}\langle\texttt{wyrażenie}\rangle\underline{\texttt{)}}.$$

  Pierwszy z identyfikatorów to nazwa definiowanej procedury, a reszta, to jej *parametry formalne*. Wyrażenie zaś to *treść* procedury.

  Proste przykłady.

  Musimy rozszerzyć nasz model obliczeniowy o obliczanie wartości kombinacji, których operatorami są procedury.

  - Aby zastosować procedurę złożoną do argumentów, należy obliczyć wartość treści procedury w środowisku, w którym (chwilowo) argumentom formalnym przyporządkowano wartości odpowiadających im argumentów.

- Procedura może być wartością wyrażenia — lambda-abstrakcja.

  $$\langle\texttt{wyrażenie}\rangle ::= \underline{\texttt{(lambda}}\ \underline{\texttt{(}}\langle\texttt{identyfikator}\rangle^{+}\underline{\texttt{)}}\ \langle\texttt{wyrażenie}\rangle\underline{\texttt{)}}.$$

  Identyfikatory to argumenty formalne. Tak zdefiniowana procedura nie ma nazwy. Proste przykłady łączące `define` i lambda-abstrakcję.

- Procedury mogą być argumentami i wynikami procedur. O tym będzie później.

Definicje stałych postaci

$$\underline{\texttt{(define}}\ \underline{\texttt{(}}\langle\texttt{nazwa}\rangle\ \langle\texttt{parametry}\rangle\underline{\texttt{)}}\ \langle\texttt{treść}\rangle\underline{\texttt{)}}$$

jest równoważna:

$$\underline{\texttt{(define}}\ \langle\texttt{nazwa}\rangle\ \underline{\texttt{(lambda}}\ \underline{\texttt{(}}\langle\texttt{parametry}\rangle\underline{\texttt{)}}\ \langle\texttt{treść}\rangle\underline{\texttt{))}}$$

Tak więc druga z przedstawionych postaci `define` jest tylko „lukrem syntaktycznym”.

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

Definicje procedur mogą być rekurencyjne, tzn. w definicji procedury może występować odwołanie do niej samej.

2.4 Wyrażenia warunkowe.

Wartości logiczne `true` i `false`, `and`, `or`, `not`. Zasady obliczania tych operatorów i ich leniwość. Ze względu na tę leniwość są to formy specjalne.

Wyrażenie warunkowe:

$$\langle\texttt{wyrażenie}\rangle ::= \underline{\texttt{(cond}}\ \{\underline{\texttt{(}}\langle\texttt{wyrażenie}\rangle\ \langle\texttt{wyrażenie}\rangle\underline{\texttt{)}}\}^{+}\underline{\texttt{)}}$$

Wyrażenie `cond` zawiera listę klauzul. Każda z klauzul to para wyrażeń. Pierwsze wyrażenie w parze to warunek, a druga to wartość. `Cond` nie jest zwykłą procedurą, jest to forma specjalna. Jest ona obliczana w następujący sposób:

- Obliczane są wartości warunków kolejnych klauzul, aż do napotkania prawdziwego warunku.
- Wartością całego wyrażenia jest wartość pierwszej klauzuli, której warunek był spełniony.
- Jeżeli żadna klauzula nie jest spełniona, to wartością całego wyrażenia jest `false`.

`Cond` jest obliczane w sposób *leniwy*, tzn. w celu obliczenia wartości wyrażenia nie są obliczane wartości wszystkich podwyrażeń. Dlatego też nie można zdefiniować `cond` jako procedury.

Zamiast pisać `(true` w`)` możemy pisać `(else` w`)`

Przykład:

```
(define (sig x)
  (cond
    ((<x 0) (-1))
    ((>x 0) 1)
    (else 0)))
```

Często chcemy sprawdzić tylko jeden warunek i zależnie od jego spełnienia bądź nie obliczyć jedno z dwóch podwyrażeń. Możemy w tym celu używać wyrażenia `if` postaci

$$\langle\texttt{wyrażenie}\rangle ::= \underline{\texttt{(if}}\ \langle\texttt{wyrażenie}\rangle_1\ \langle\texttt{wyrażenie}\rangle_2\ \langle\texttt{wyrażenie}\rangle_3\underline{\texttt{)}}$$

Jest ono równoważne:

$$\underline{\texttt{(cond}}\ \underline{\texttt{(}}\langle\texttt{wyrażenie}\rangle_1\ \langle\texttt{wyrażenie}\rangle_2\underline{\texttt{)}}\ \underline{\texttt{(else}}\ \langle\texttt{wyrażenie}\rangle_3\underline{\texttt{))}}$$

Przykład:

```
(if (= x 0) 1 (* 2 x))
```

Przykład: `If` zapisany jako procedura nie działa (np. dzielenie przez 0, ale też rekurencja — patrz dalej).

Ćwiczenia

Proste programiki operujące liczbami całkowitymi, bez rekurencji ogonowej:

- silnia,
- liczby Fibonacciego,
- sprawdź, czy liczba jest pierwsza,
- sprawdzenie podzielności przez 9 metodą sumowania cyfr: liczba -> suma cyfr, powtarzać aż do uzyskania liczby jednocyfrowej,
- odwrócić liczbę ($1234 \to 4321$),
- sprawdzenie, czy liczba dzieli się przez 11: sumujemy cyfry liczby znajdujące się na parzystych pozycjach, oraz te na nieparzystych pozycjach, różnica tych dwóch reszt przystaje modulo 11 do wyjściowej liczby; powtarzać aż do uzyskania liczby jednocyfrowej,
- kodowanie par liczb całkowitych jako liczby całkowite.

3 Wyliczanie procesów — proces wyliczania

3.1 Przypomnienie

Do tej pory został przedstawiony następujący model:

- Dla wyrażeń budujemy ich drzewa składni abstrakcyjnej i dekorujemy je wartościami, od liści do korzenia.
- Środowisko — rodzaj słownika przyporządkowującego nazwom wartości. Definicja wartości to rozszerzenie słownika.

Model ten nie przedstawia jednak dobrze działania procedur oraz definicji lokalnych.

3.2 Definicje lokalne

Definicje lokalne mają dokładnie taką samą treść jak globalne, ale są umieszczone w treści innych definicji, zwykle definicji procedur. Ich zakres jest ograniczony do definicji, w której się znajdują.

Przykład:

```
(define (pitagoras x y)
  (define (square x) (* x x))
  (+ (square x) (square y))
)
```

Jak opisać wzajemne relacje między argumentami procedur `pitagoras` i `square`? Czy oba `x`-y to ten sam `x`? Aby to opisać musimy sprecyzować strukturę środowiska.

3.3 Środowisko i ramki.

Podział środowiska na ciąg ramek połączonych wskaźnikami w listę jednokierunkową. Poszukując w środowisku wartości stałej przeglądamy kolejne ramki. Definiowanie nowych wartości powoduje dodanie ich do ostatniej ramki, lub zmodyfikowanie definicji znajdującej się w ostatniej ramce. Każda ramka reprezentuje pewne środowisko. Aktualne środowisko jest reprezentowane przez ostatnio dodaną ramkę. Środowisko, do którego prowadzi wskaźnik od danej ramki nazywamy środowiskiem otaczającym.

3.4 Model obliczeń

`Define` wstawia definiowany symbol do pierwszej ramki, lub jeśli już on tam istnieje, to zmienia jego znaczenie.

Wartość procedury to trójka: nazwy argumentów, treść i wskaźnik do ramki reprezentującej środowisko, w którym zdefiniowano procedurę.

3.4.1 Wyliczenie kombinacji

Obliczenie wartości kombinacji polega na:

- wyliczeniu wszystkich elementów kombinacji (wliczając w to operator),
- stworzeniu nowej ramki, dla której środowiskiem otaczającym jest środowisko, w którym zdefiniowano procedurę, w której nazwom parametrów formalnych przyporządkowano wartości argumentów,
- wyliczeniu w takim środowisku wartości wyrażenia stanowiącego treść.

Przykład: Rozrysować następujące obliczenie:

```
(define x 5)
(define (square x) (* x x))
(define (pitagoras x y)
  (+ (square x) (square y))
)
(pitagoras (+ x 1) (* x 2))
```

Przykład:

```
((lambda (x) x) 7)
```

Przykład: Rozrysować następujące obliczenie:

```
(define (pitagoras x y)
  (define (square x) (* x x))
  (+ (square x) (square y))
)
(pitagoras 3 4)
```

3.4.2 Rekurencja

Procedury rekurencyjne — wskaźnik tworzący wartość procedury wskazuje na środowisko, do którego trafia definicja procedury.

Przykład:

- jak się wyliczy silnia 3,
- jak zapisać sumę $1 + 2 + .. + n$.

3.4.3 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 aktualną ramkę zmieniając jedynie wartości argumentów. Przykład: silnia z akumulatorem.

3.4.4 Odśmiecanie

Ramka zawierająca lokalne definicje nie musi stawać się od razu zbędna, gdyż mogą na nią wskazywać wartości proceduralne. Zbędne ramki są usuwane w miarę potrzeb — zbędne to takie, na które nic nie wskazuje lub, do których nie można dotrzeć z aktualnego środowiska poprzez wskaźniki. Ramki nie tworzą listy, ale drzewo (biorąc pod uwagę wskaźniki na otaczające środowisko).

Przykład:

```
(define (f x)
  (define (g y) (+ x y))
  g)
((f 4) 2)
```

Przykład:

```
(define (g x)
  (define a (* x x))
  (lambda (y) (+ y a)))
((g 4) 2)
```

3.5 Wyrażenia `let`

Do tworzenia definicji lokalnych wystarczy sam mechanizm procedur — wartości parametrów są lokalne dla treści procedury.

Przykład:

```
(define (pitagoras x y)
  ((lambda (square) (+ (square x) (square y)))
    (lambda (x) (* x x))
  )
)
```

Jest to na tyle przydatna konstrukcja, że stworzono dla niej równoważny jej lukier syntaktyczny postaci:

$$\langle\texttt{wyrażenie}\rangle ::= \underline{\texttt{(let (}}\ \{\underline{\texttt{(}}\langle\texttt{identyfikator}\rangle\ \langle\texttt{wyrażenie}\rangle\underline{\texttt{)}}\}^{+}\underline{\texttt{)}}\ \langle\texttt{wyrażenie}\rangle\underline{\texttt{)}}.$$

Wyrażenie postaci:

`(let (`
  $(x_1 w_1)$
      $\vdots$
  $(x_n w_n)$`)`
  t
`)`

jest równoważne:

`((lambda (`$x_1 \ldots x_n$`)` t`)` $w_1 \ldots w_n$`)`

Przykład: Poprzedni przykład zapisany w tej postaci to:

```
(define (pitagoras x y)
  (let
    ((square (lambda (x) (* x x))))
    (+ (square x) (square y))
  )
)
```

Ewaluacja wyrażeń `let`:

- tworzona jest ramka zawierająca definicję symboli lokalnych (otaczającym ją środowiskiem jest aktualne środowisko),
- w tak powstałym środowisku wyliczana jest wartość wyrażenia,
- po czym aktualnym środowiskiem jest to, co poprzednio.

Przykład:

```
(define (f x)
  (let ((y (-x 4)))
    (let ((z (* 2 y)))
      (+ x y z)
    )
  )
)
(f 2)
```

3.5.1 Porównanie `define` i `let`

Zauważmy, że `let` zawsze tworzy nową ramkę, a więc może **`przysłonić`** definicje ze środowiska, ale ich nie zmieni. Ponadto wyrażenie `let` ogranicza zakres definicji lokalnych jak to tylko możliwe. Z drugiej strony, `define` pozwala definiować procedury lokalne wzajemnie rekurencyjne. Dlatego też nie będziemy wchodzić przesadnie w niuanse techniczne i będziemy używać `define` do definiowania procedur lokalnych, natomiast wyrażeń `let` będziemy używać do definiowania kolejno od siebie zależnych wielkości.

4 Dekompozycja problemu, weryfikacja rozwiązania

4.1 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.

$$\forall x : x > 0 => ....$$

  ```
  (define (f x) (if (> x 0) (...)  (...)
  ```

- 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 wstępny muszą istnieć wyniki spełniające warunek końcowy.
- To, że program spełnia specyfikację można wyrazić formułą:

$$\text{warunek wstępny} \Rightarrow \text{warunek końcowy(wynik programu)}$$

4.2 Euklides (wersja z odejmowaniem)

Specyfikacja: dla $n, m > 0$ mamy $(\texttt{nwd}mn) = NWD(m, n)$. implementacja:

```
(define (nwd x y)
  (cond
    ((= x y) x)
    ((>x y) (nwd (-x y) y))
    (else (nwd x (-y x)))
  )
)
```

Nowy rodzaj nieokreśloności — zapętlenie. Pojęcia częściowej i całkowitej poprawności — w przypadku częściowej poprawności implementacja może być mniej określona niż specyfikacja.

Jak weryfikować rekurencję:

- funkcja miary,
- dla danych spełniających warunek wstępny wartość funkcji miary jest nieujemna,
- indukcyjnie (po wartościach funkcji miary) dowodzimy, że:

– wywołania rekurencyjne mają miejsce tylko dla mniejszych wartości funkcji miary i spełniają warunki początkowe,

– przy założeniu, że wyniki wywołań rekurencyjnych spełniają warunek końcowy, wyniki spełniają również warunek końcowy.

Część dowodu dotycząca funkcji miary dowodzi własności stopu programu, tzn. że wywołania rekurencyjne nie zapętlą się.

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 $\mathrm{nwd}(x, y) = \mathrm{nwd}(x - y, y)$. Dla $x < y$ analogicznie. □

Zauważmy, że program będzie dawał poprawne wyniki również gdy jeden z parametrów będzie zerem.

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

4.3 Metoda Newtona liczenia pierwiastków

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

Specyfikacja: chcemy mieć funkcję, której wartość podniesiona do kwadratu jest bliska wartości argumentu (dla nieujemnych argumentów), tzn. różni się od niej np. o co najwyżej 0.0001.

Metoda: zaczynam od pewnej wartości i iteruję, aż wynik będzie dobry.

```
(define (sqrt x) (sqrt-iter (guess x) x))
(define (sqrt-iter g x)
  (if (good? g x) g (sqrt-iter (improve g x) x))
)
```

Pierwszy kandydat to 1.

```
(define (guess x) 1.0)
```

Jeśli `g` nie jest dobre, to szukana liczba leży gdzieś między `g`, a `x/g`. Dobrym kandydatem jest więc średnia arytmetyczna tych liczb.

```
(define (improve g x) (average g (/ x g)))
(define (average x y) (/ (+ x y) 2.0))
```

Przybliżenie jest dobre, gdy podniesione do kwadratu nie różni się od `x` o nie więcej niż 0.0001.

```
(define (good? g x) (<(abs (-(* g g) x)) epsilon))
(define (abs x) (if (<x 0) (-x) x))
(define epsilon 0.00001)
```

Weryfikacja: poprawność częściowa jest oczywista, ale jak pokazać własność stopu?

Niech nasze przybliżenie będzie postaci $\sqrt{x} + e$. Wówczas błąd kolejnego przybliżenia wynosi:

$$\frac{\sqrt{x} + e + \frac{x}{\sqrt{x}+e}}{2} - \sqrt{x} = \frac{\sqrt{x} + e + \frac{x}{\sqrt{x}+e} - 2\sqrt{x}}{2} = \frac{(-\sqrt{x} + e)(\sqrt{x} + e) + x}{2(\sqrt{x} + e)} =$$

$$\frac{-x+e^2+x}{2(\sqrt{x}+e)} = \frac{e^2}{2(\sqrt{x}+e)} = \frac{1}{2}e\frac{1}{1+\frac{\sqrt{x}}{e}}$$

Zakładamy, że nasze przybliżenie jest dodatnie. 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 zerowy, to wychodzimy oczywiście z pętli. Tak więc wszystkie przybliżenia są dodatnie, a tylko w pierwszym kroku błąd może być ujemny.

Możemy jako funkcję miary przyjąć:

$$\max\left(\left\lceil \log_2 \frac{g-\sqrt{x}}{0.0001}\right\rceil, 0\right)$$

(A dla błędów ujemnych? ....uzupełnić)

Jak widać dowód własności stopu jest trudniejszy niż dowód poprawności.

4.4 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`,
- średnia `average`,
- wartość bezwzględna `abs`.

Jak dzielić problem na podproblemy? To trudna umiejętność, która istotnie wpływa na to czy będziemy dobrymi informatykami. 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 przystępujemy do ich rozwiązywania. Jest to tzw. zasada *pobożnych życzeń*. W ten sposób rozwiązujemy problem *top-down* — od ogółu do szczegółów.

Kilka innych zasad:

- Black-box approach: Polegamy na tym, że dana wartość/funkcja 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 rozwiązania różnych problemów są od siebie niezależne i nie kolidują ze sobą. Na raz możemy zajmować się tylko jednym problemem.

Laboratorium

Zaprogramuj ciekawsze programy operujące na liczbach oraz procedury wyższych rzędów i przećwicz ich działanie.

Ćwiczenia

Proste programiki operujące na liczbach całkowitych i procedurach (tym razem zaczynamy zwracać uwagę na rekurencję ogonową):

- silnia (z akumulatorem i rekurencją ogonową), wykaż poprawność rozwiązania (wymaga wyspecyfikowania procedury pomocniczej),
- liczby Fibonacciego (z akumulatorem i rekurencją ogonową), wykaż poprawność rozwiązania (wymaga wyspecyfikowania procedury pomocniczej),
- ew. inne programy z poprzednich zajęć z rekurencją ogonową,
- $f(x) \to 2x$,
- $g(x) \to f(f(x))$,
- $powt(f, x) \to f(f(x))$,
- $id(x) \to x$,
- $zloz(f, g) \to f \circ g$,
- $iter(n, f) \to f^n$,

5 Procedury wyższych rzędów jako abstrakcje konstrukcji programistycznych

W punkcie tym przedstawimy na przykładach, w jaki sposób można używać procedur, których argumentami i/lub wynikiem są procedury jako abstrakcji powtarzających się schematów postępowania.

5.1 Sumy częściowe szeregów

Przykład — szereg (kolejnych liczb naturalnych, kwadratów, $\frac{1}{(4i-3)(4i-1)}$ zbieżny do $\frac{\pi}{8}$. Istota szeregu jako funkcja wyższego rzędu, sumowane elementy jako funkcje liczb całkowitych. Procedura sumująca n pierwszych elementów danego szeregu (podać specyfikację `sum`):

```
(define (szereg f n)
  (define (sum a n)
    (if (= n 0)
      a
      (sum (+ a (f n)) (-n 1))))
  (sum 0 n))
(define (szereg-pi-8 n)
  (szereg
    (lambda (i) (/ 1 (* (-(* 4 i) 3) (-(* 4 i) 1))))
    n))
(define pi (* 8 (szereg-pi-8 1000)))
```

Procedura `szereg` może służyć do obliczania sum częściowych dowolnych szeregów. Przypomnienie hasła „abstrakcja proceduralna” — tutaj abstrahujemy nie procedury pomocnicze, ale wspólny schemat „procedury głównej”, która operuje na różnych procedurach lokalnych.

5.2 Szukanie zer

Mamy daną ciągłą funkcję i staramy się przybliżyć punkt, w którym funkcja ta przecina 0. Przedstawimy dwie takie metody:

- 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 polega na narysowaniu stycznej do wykresu funkcji (różniczkowalnej) w danym punkcie i zbadaniu, gdzie styczna ta przecina oś X. Dla rozważanych przez nas funkcji proces ten zbiega do zera funkcji.

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

```
(define (iteruj początek popraw czy-dobre? wynik)
  (if (czy-dobre? początek)
    (wynik początek)
    (iteruj (popraw początek) popraw czy-dobre? wynik)))
```

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

- `początek`
- `popraw`
- `czy-dobre?`
- `wynik`

5.3 Metoda przez bisekcję

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

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

```
(define (bisekcja f l p)
  (let ((wartość-l (f l))
        (wartość-p (f p)))
    (cond ((and (ujemne? wartość-l) (dodatnie? wartość-p))
           (szukaj f l p))
          ((and (dodatnie? wartość-l) (ujemne? wartość-p))
           (szukaj f p l))
          (else
            (error "Wartości funkcji nie mają przeciwnych znaków:ł p))
    )
  )
)
(define (dodatnie? x) (>x 0))
(define (ujemne? x) (<x 0))
```

Wykorzystujemy procedurę `iter`.

```
(define (szukaj f l p)
  (iteruj
    (zapakuj f l p)
    popraw-bis
    czy-dobre-bis?
    rozpakuj-l))
```

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.

```
(define (czy-dobre-bis? x)
  (<(abs ((rozpakuj-f x) (rozpakuj-l x))) epsilon))
(define (popraw-bis x)
  (let (
    (f (rozpakuj-f x))
```

```
    (l (rozpakuj-l x))
    (p (rozpakuj-p x)))
    (let (
      (środek (average l p)))
      (if (dodatnie (f środek))
        (zapakuj f l środek)
        (zapakuj f środek p)))))
```

Pozostały nam do zdefiniowania procedury: `zapakuj`, `rozpakuj-f`, `rozpakuj-l` i `rozpakuj-p`, które nie są niczym innym jak włożeniem w produkt kartezjański i rzutowaniami na współrzędne. Ich realizacja jest trywialna, odłożymy ją jednak do chwili, gdy poznamy *pary* i *struktury listowe*.

5.4 Metoda Newtona

Zakładamy, że dana funkcja jest różniczkowalna. Pomijamy kwestie stopu i dzielenia przez 0.

Metoda Newtona, nazywana też metodą stycznych, polega na przybliżaniu zera funkcji w następujący sposób: w punkcie będącym aktualnym przybliżeniem rysujemy styczną do wykresu funkcji i kolejnym przybliżeniem jest punkt przecięcia stycznej z osią X. Zauważmy, że jeżeli naszym przybliżeniem zera jest x, to styczna do wykresu funkcji przecina oś X w punkcie $x - \frac{(f\ x)}{(f'\ x)}$.

```
(define (newton f x)
  (let (
    (p (pochodna f)))
    (define (czy-dobre-newton? x)
      (<(abs (f x)) epsilon))
    (define (popraw-newton x)
      (-x (/ (f x) (p x))))
    (iteruj x popraw-newton czy-dobre-newton? id)))
(define (id x) x)
```

Zwróćmy uwagę na to, że procedura `popraw-newton` jest lokalna i ma dostęp do funkcji `f`. Pozostało nam jeszcze napisanie procedury `pochodna`.

Metoda Newtona liczenia pierwiastków kwadratowych 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

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

Zaczynając w punkcie 1 przybliżamy $\sqrt{a}$.

```
(define (sqrt a)
  (newton (lambda (x) (-(square x) a)) 1.0))
```

5.5 Różniczkowanie funkcji

Zdefiniujmy najpierw różniczkę:

```
(define (różniczka f x dx)
  (/ (-(f (+ x dx)) (f x)) dx))
```

Wówczas pochodną możemy przybliżyć następująco:

```
(define (pochodna f)
  (lambda (x) (różniczka f x epsilon)))
```

Zwróćmy uwagę, że `pochodna` jest procedurą jednoargumentową i wynikiem `(pochodna f)` jest funkcja będąca pochodną `f`.

5.6 Punkty stałe funkcji

Punkt stały funkcji f, to taki x, że $f(x) = x$. W przypadku niektórych funkcji (i określonych x-ów) ciąg $x, f(x), f^2(x), f^3(x), \ldots$ jest zbieżny do pewnego punktu stałego f. Możemy zaimplementować tę metodę:

```
(define (punkt-stały f x)
  (define (blisko? x1 x2)
    (<(abs (-x1 x2)) epsilon))
  (define (iteruj x)
    (let ((y (f x)))
      (if (blisko? x y) y (iteruj y))))
  (iteruj x))
```

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

```
(punkt-stały cos 1.0)
0.739082...
```

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

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

```
(punkt-stały (lambda (y) (/ x y)) 1.0)
```

nie jest zbieżne:

$$1 \to \frac{x}{1} = x \to \frac{x}{x} = 1 \to \ldots$$

W takich przypadkach czasami pomaga „wytłumienie” oscylacji takiego ciągu przez jego uśrednienie.

```
(define (uśrednienie f)
  (lambda (x) (average x (f x))))
(define (sqrt x)
  (punkt-stały (uśrednienie (lambda (y) (/ x y))) 1.0))
```

W ten sposób uzyskujemy po raz kolejny metodę pierwiastkowania Newtona.

Ćwiczenia

- Potęgowanie funkcji z akumulatorem.
- Szybsze potęgowanie funkcji z akumulatorem.
- Liczenie pierwiastka przez bisekcję z dowodem poprawności. Uwaga na warunki brzegowe.
- 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.
- Punktem stałym funkcji $y \to \frac{x}{y^{n-1}}$ jest $\sqrt[n]{x}$. Zaimplementuj obliczanie n-tego pierwiastka z x za pomocą obliczania punktu stałego i tłumienia przez uśrednianie. Uwaga: ile razy należy stosować tłumienie w zależności od n?

6 Analiza kosztów

6.1 Rzędy funkcji

$$f(x) = \Theta(g(x)) \Leftrightarrow \exists_{c_1,c_2 \in real, c_1,c_2>0, n_0 \in nat} \forall_{n \in nat, n \geq n_0} c_1 g(n) \leq f(n) \leq c_2 g(n)$$

$$f(x) = O(g(x)) \Leftrightarrow \exists_{c \in real, c>0, n_0 \in nat} \forall_{n \in nat, n \geq n_0} 0 \leq f(n) \leq cg(n)$$

$$f(x) = \Omega(g(x)) \Leftrightarrow \exists_{c \in real, c>0, n_0 \in nat} \forall_{n \in nat, n \geq n_0} 0 \leq cg(n) \leq f(n)$$

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:

- $n^2 + 100\,000$,
- $0.000001 n^{\frac{1}{2}}$,
- 4,
- 2^{2n},
- $\log_2 n$,
- $2\sqrt{4n}$,
- $\ln(10n)$,
- 4^n,
- 10^n,
- 0.

6.2 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 $\varphi : D \to N$.

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.

$$\vec{\varphi} \circ \vec{\mu}^{-1} : N \to \mathcal{P}(N)$$

Wybierając koszt najmniejszy lub największy mówimy o koszcie pesymistycznym lub optymistycznym.

$$\max \circ\, \vec{\varphi} \circ \vec{\mu}^{-1} : N \to N$$

$$\min \circ\, \vec{\varphi} \circ \vec{\mu}^{-1} : N \to N$$

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.

6.3 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,
- obliczenie kombinacji — koszt obliczenia elementów kombinacji (w tym operatora) + koszt wyliczenia treści operatora (w przypadku operatorów wbudowanych koszt ich obliczania wynosi 1, chyba że powiedziane jest inaczej) + liczba elementów kombinacji,
- wyrażenie `cond` — koszt obliczania warunków kolejnych klauzul + koszt obliczenia następnika odpowiedniej klauzuli,
- `if` — koszt obliczenia warunku i zależnie od wyniku koszt obliczenia odpowiedniej części,
- wyrażenie `lambda` — 1 (uwaga: koszt ujawnia się dopiero przy zastosowaniu do argumentów),
- definicje `define` — koszt obliczenia definiowanej wartości + 1,
- wyrażenie `let` — zgodnie z rozwinięziem do `lambda`-wyrażenia i kombinacji, czyli koszt wyliczenia definiowanych lokalnie wartości + wyliczenie właściwego wyrażenia + liczba lokalnie definiowanych wartości + 1,
- rekurencja — równanie rekurencyjne na kosztach.

Przykład:

- nierekurencyjny przykład z `let`:

```
(define f
  (define (g x) (* 3 x))
  (lambda (x) (g (* 4 x))))
(f 5)
```

- silnia,

6.4 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 lista, to potrzebne są również jej składowe.

Wartości mogą być współdzielone, tzn. na tę samą wartość może wskazywać kilka wskaźników. Wartości:

- liczby mają rozmiar 1,
- lista — to będzie omówione później,
- rozmiar wartości funkcyjnych — 1 + liczba symboli w wyrażeniu.

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. Zasada 90%–10%. Ponieważ interesuje nas tylko rząd kosztu, możemy stosować uproszczenia już w trakcie jego wyliczania:

- 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.

6.5 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:

```
(define (potęga b n)
  (if (= n 0) 1 (* b (potęga b (-n 1)))))
```

Liczba kroków jest równa `n`, każdy krok ma stały koszt i wymaga stałej ilości pamięci. Koszt czasowy i pamięciowy $\Theta(n)$.

Koszt pamięciowy możemy polepszyć stosując rekurencję ogonową.

```
(define (potęga b n)
  (define (pot a b n)
    (if (= n 0) a (pot (* a b) b (-n 1))))
    (pot 1 b n))
```

$$(\texttt{pot}\ abn) = ab^n$$

Koszt czasowy pozostaje jednak bez zmian.

Możemy jednak skorzystać z innego wzoru na potęgowanie:

$$\begin{array}{rcl} b^0 & = & 1 \\ b^{2n} & = & \left(b^2\right)^n \\ b^{2n+1} & = & b \cdot b^{2n} \end{array}$$

który zapisujemy jako:

```
(define (potęga b n)
  (define (pot a b n)
    (cond
      ((= n 0) a)
      ((parzyste? n) (pot a (* b b) (/ n 2)))
      (else (pot (* a b) b (-n 1)))))
  (pot 1 b n))
```

Mamy tutaj rekurencję ogonową, więc koszt pamięciowy jest stały. Jaka jest jednak liczba kroków? Można pokazać przez indukcję, że jeżeli nasz algorytm wymaga k kroków (dla $k \geq 2$), to $2^{k/2} - 1 \leq n \leq 2^{k-2}$. Stąd, złożoność czasowa jest rzędu $\Theta(\log n)$.

6.6 Przykład: algorytm Euklidesa przez odejmowanie

```
(define (nwd a b)
  (cond
    ((= a b) a)
    ((>a b) (nwd (-a b) b))
    (else (nwd a (-b a)))))
```

Dla $n, m > 0$ mamy $(\texttt{nwd}mn) = NWD(m, n)$.

Jaka jest złożoność czasowa (ze względu na $a + b$)? Liczba wykonywanych kroków może być liniowa, np. dla $(\texttt{nwd}\ a\ 1)$, czyli $T(n) = \Omega(n)$. Gorsza nie będzie, bo z każdym krokiem maleje wartość $a + b$, $T(n) = O(n)$, czyli $T(n) = \Theta(n)$.

Mamy tu do czynienia z rekurencją ogonową, więc złożoność pamięciowa jest stała.

6.7 Przykład: algorytm Euklidesa przez dzielenie

```
(define (nwd a b)
  (define (e a b)
    (if (= b 0) a (e b (modulo a b))))
  (if (>a b) (e a b) (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 Lame'a Oznaczmy przez (a_i, b_i) pary wartości a i b, dla których powyższy algorytm wykonuje i kroków. Wówczas $b_i \geq fib_{i-1}$. Dowód indukcyjny.

1. jeśli jeden krok, to $b = 0$,
2. jeśli dwa kroki, to $a = kb$, $b \geq 1$,
3. jeśli więcej kroków, to $(a_{k+1}, b_{k+1}) \to (a_k, b_k) \to (a_{k-1}, b_{k-1})$, to $a_k = b_{k+1}$, $a_{k-1} = b_k$, $b_{k-1} = a_k \bmod b_k$, 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:

$$fib_n \approx \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

czyli koszt czasowy $T(n) = O(\log n)$. Z drugiej strony mamy

$$fib_{n+1} \bmod fib_n = (fib_n + fib_{n-1}) \bmod fib_n = fib_{n-1}$$

czyli dla $a = fib_{n+1}$ i $b = fib_n$ algorytm wykonuje n kroków. Stąd $T(n) = \Theta(\log n)$.

Koszt pamięciowy ze względu na rekurencję ogonową wynosi $\Theta(1)$.

6.8 Przykład: algorytm Euklidesa przez parzystość

Zastosowanie zasady dziel i zwyciężaj.

```
(define (nwd a b)
  (cond
    ((= a b) a)
    ((and (parzyste? a) (parzyste? b)) (* 2 (nwd (/ a 2) (/ b 2))))
    ((and (parzyste? a) (not (parzyste? b))) (nwd (/ a 2) b))
    ((and (not (parzyste? a)) (parzyste? b)) (nwd a (/ b 2)))
    (else (if (>a b) (nwd (-a b) b) (nwd a (-b a))))))
```

Algorytm ten nie wykorzystuje jedynie odejmowanie i mnożenie/dzielenie/modulo 2. Ze względu na zapis binarny liczb operacje te mają koszt O(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 $\Theta(\log\max(a,b))$. Nie mamy tu do czynienia z rekurencją ogonową. Każdy krok algorytmu może wymagać stałej ilości pamięci, stąd koszt pamięciowy wynosi $O(\log\max(a,b))$.

Jakie będą koszty algorytmu w przypadku długich liczb? (*długość liczb)

6.9 Test na liczby pierwsze

Jak sprawdzić czy dana liczba jest liczbą pierwszą? Należy sprawdzić, czy ma jakieś dzielniki, do $\sqrt{n}$.

```
(define (min-dzielnik n)
  (define (dziel k)
    (if (>(* k k) n)
      n
      (if (= (modulo n k) 0) k (dziel (+ k 1)))))
  (dziel 2))
```

Dzięki rekurencji ogonowej koszt pamięciowy jest stały. Liczba kroków nie przekracza $\sqrt{n}$, czyli złożoność czasowa jest rzędu $O(\sqrt{n})$.

6.10 Test Fermata

Oto algorytm Millera-Rabina sprawdzania, czy dana liczba jest pierwsza.

Małe twierdzenie Fermata Jeśli n jest liczbą pierwszą, a a jest dowolną dodatnią liczbą mniejszą niż n, to $a^{n-1} \equiv 1 \bmod n$.

Fakt Jeśli n ma nietrywialny pierwiastek 1, czyli istnieje takie $1 < k < n-1$, że $k^2 \equiv 1 \bmod n$, to n nie jest liczbą pierwszą.

Nasz algorytm polega na wylosowaniu liczby a i sprawdzeniu, czy spełnione jest tw. Fermata. Ponadto obliczając $a^{n-1} \bmod n$ sprawdzamy, czy nie znajdziemy nietrywialnego pierwiastka 1. Przypadki, gdy natrafiamy na nietrywialny pierwiastek 1 sygnalizujemy wynikiem równym 0.

```
(define (square x) (* x x))
(define (expmod b k n)
  (define (test x)
    (if
      (and
        (not (= x 1))
        (not (= x (-n 1)))
```

```
        (= (modulo (square x) n) 1))
      0
      x))
  (if (= k 0)
    1
    (if (parzyste? k)
      (modulo (square (test (expmod b (/ k 2) n))) n)
      (modulo (* (test (expmod b (-k 1) n)) b) n))))
```

Można pokazać, że jeżeli n jest liczbą nieparzystą i nie jest liczbą pierwszą, to przynajmniej dla połowy $1 < a < n-1$ obliczanie a^{n-1}mod n odkryje nietrywialny pierwiastek 1.

```
(define (randtest n)
  (if (parzyste? n)
    (= n 2)
    (= (expmod (+ (random (-n 3)) 2) (-n 1) n ) 1)))
```

Test ten z prawdopodobieństwem co najmniej 1/2 daje poprawną odpowiedź (a tak naprawdę z dużo większym), a z prawdopodobieństwem nie przekraczającym 1/2 uzna liczbę, która nie jest pierwsza, za pierwszą.

Gadka o algorytmach randomizacyjnych. Dwie klasy:

- Monte Carlo — złożoność zawsze dobra, ale z małym prawdopodobieństwem może dawać złe wyniki,
- Las Vegas — zawsze daje dobre wyniki, ale z małym prawdopodobieństwem działa dłużej; średnia złożoność musi być OK.

Jęśli 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 taki sam jest koszt całego algorytmu.

Ćwiczenia

- Algorytm mnożenia rosyjskich chłopów.
- Różne sposoby liczenia liczb Fibbonacciego i ich koszty:
  - nieefektywnie ($Fib_n \approx \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}}$),

    ```
    (define (fib n)
      (if (<n 3) 1 (+ (fib (-n 2)) (fib (-n 1)))))
    ```
  - z parami i rekurencją ogonową,

    ```
    (define (fib n)
      (define (f a b n)
        (if (>n 1) (f b (+ a b) (-n 1)) b))
      (f 0 1 n));;
    ```
  - w koszcie czasowym rzędu$\Theta(\log n)$ i stałym pamięciowym (nie korzystając ze wzoru na n-tą liczbę Fibonacciego).
- Chiński łańcuch (przynieść) — liczba ruchów potrzebnych do zdjęcia.

Laboratorium

- Zaprogramować znajdowanie zer funkcji metoda Newtona, $x \to x - \frac{f(x)}{f'(x)}$, z wbudowanym przybliżeniem pochodnej.
- Obliczanie pierwiastka jako instancja metody Newtona. Przetestować.
- Znajdowanie punktu stałego funkcji.
- Tłumienie przez uśrednianie.
- Obliczanie $\sqrt[n]{a}$ jako punkt stały $x \to \frac{a}{x^{n-1}}$. Ile razy należy tę funkcję tłumić (w zależności od n).

7 Struktury danych

Dotąd, tworzone przez nas abstrakcje to wyłącznie procedury. Dzisiaj zajmiemy się tworzeniem złożonych obiektów danych. Zajmiemy się dokładniej metodyką tworzenia nowych *typów* danych. Co to jest typ? Jest to zbiór wartości wraz z zestawem podstawowych operacji na tych wartościach. Uwaga: Scheme nie sprawdza typów danych — jest to pojęcie z wyższego poziomu, dotyczące projektowanych i konstruowanych programów.

7.1 Budowanie abstrakcji za pomocą danych

Na poprzednich wykładach pojawiły się już pary. Używaliśmy procedur `zapakuj`, `rozpakuj-p` i `rozpakuj-f`. Nie podaliśmy jednak w jaki sposób te procedury są zaimplementowane. Ustaliliśmy więc pewnien `interfejs` składający się z operacji na nowym typie danych. Operacje takie zwykle można podzielić na:

- konstruktory — procedury tworzące złożone wartości struktury danych z innych, prostszych wartości, np. `zapakuj`,
- selektory — procedury wydobywające poszczególne wartości składowe lub cechy złożonej struktury danych, np. `rozpakuj`,
- modyfikatory — procedury tworzące nowe złożone wartości z innych złożonych wartości.

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 urzytkownik typu, a konkretne to sposób ich implementacji. O funkcji abstrakcji. Diagram.

Czasami odpowiednio dobrane konstruktory i selektory mogą tworzyć dodatkową abstrakcję, za pomocą której są zaimplementowane modyfikatory. Wówczas implementacja modyfikatorów korzysta wyłącznie z wartości abstrakcyjnych i nie jest wrażliwa na zmianę 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.

programy korzystające ze złożonej struktury danych	
	modyfikatory
implementacja modyfikatorów	
	konstruktory i selektory
implementacja struktury danych	

7.2 Liczby wymierne

7.2.1 Abstrakcja danych

Spróbujmy zaimplementować zgodnie z opisanym podejściem pakiet arytmetyki liczb wymiernych. Będziemy używać jednego konstruktora:

- (`ułamek` $l m$) tworzący ułamek $\frac{l}{m}$ przy założeniu, że $m \neq 0$,

i trzech selektorów:

- (`licznik` q),
- (`mianownik` q),
- (`równe` q_1 q_2).

Wymagamy przy tym, aby zachodziły następujące równości:

$$\forall_{l,m\in\mathcal{N},m\neq 0}\frac{(\texttt{licznik}\ (\texttt{ułamek}\ l\ m))}{(\texttt{mianownik}\ (\texttt{ułamek}\ l\ m))} = \frac{l}{m}$$

$$\forall_{q_1,q_2}(\texttt{równe}\ q_1\ q_2) \equiv (\texttt{licznik}\ q_1)\cdot(\texttt{mianownik}\ q_2) = (\texttt{licznik}\ q_2)\cdot(\texttt{mianownik}\ q_1)$$

Dowolna implementacja spełniająca te warunki będzie nas satysfakcjonować. Użytkownik pakietu może polegać na *specyfikacji*, ale nie może nic zakładać o jej *implementacji*. Zgodnie z techniką pobożnych życzeń, odkładamy implementację tych trzech operacji na później.

7.2.2 Modyfikatory

Modyfikatory to operacje arytmetyczne. Możemy je zaimplementować zgodnie z następującymi tożsamościami:

$$\frac{l_1}{m_1}+\frac{l_2}{m_2} = \frac{l_1m_2+l_2m_1}{m_1m_2}$$
$$\frac{l_1}{m_1}-\frac{l_2}{m_2} = \frac{l_1m_2-l_2m_1}{m_1m_2}$$
$$\frac{l_1}{m_1}\cdot\frac{l_2}{m_2} = \frac{l_1l_2}{m_1m_2}$$
$$\frac{l_1/m_1}{l_2/m_2} = \frac{l_1m_2}{l_2m_1}$$

```
(define (add-wym x y)
  (ułamek (+ (* (licznik x) (mianownik y))
             (* (licznik y) (mianownik x)))
          (* (mianownik x) (mianownik y))))

(define (sub-wym x y)
  (ułamek (-(* (licznik x) (mianownik y))
             (* (licznik y) (mianownik x)))
          (* (mianownik x) (mianownik y))))

(define (mul-wym x y)
  (ułamek (* (licznik x) (licznik y))
          (* (mianownik x) (mianownik y))))

(define (div-wym x y)
  (ułamek (* (licznik x) (mianownik y))
          (* (mianownik x) (licznik y))))

(define (equal-wym? x y)
  (= (* (licznik x) (mianownik y))
     (* (licznik y) (mianownik x))))
```

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

Oprócz modyfikatorów możemy powyżej bariery astrakcji oddzielającej wartości konkretne od abstrakcyjnych zaimplementować selektor `równe`.

```
(define (równe q1 q2)
  (=
    (* (licznik q1) (mianownik q2))
    (* (licznik q2) (mianownik q2))))
```

7.3 Pary

Do zaimplementowania konstruktorów i selektorów pakietu liczb wymiernych będziemy potrzebowali par. Pary to elementy produktu kartezjańskiego. Tworzymy je z pomocą *konstruktora* `cons`, `(cons x y)` tworzy parę (x, y). Do rzutowania pary na współrzędne służą *selektory* `car` i `cdr`.

```
(define x (cons 1 2))
(car x)
1
(cdr x)
2
```

Realizacja wskaźnikowa par. Można też tworzyć złożone pary, tzw. struktury listowe.

```
(define x (cons 1 2))
(define y (cons 3 4))
(define z (cons x y))
(car (car z))
1
(car (cdr z))
3
```

7.3.1 Reprezentacja liczb wymiernych

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

- określić 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, mainownik). Wartości konkretne to wszystkie takie pary (l, m), w których $m \neq 0$. Funkcja abstrakcji parze (l, m) przyporządkowuje ułamek $\frac{l}{m}$. Operacje `ułamek`, `mianownik` i `licznik` muszą spełniać specyfikacje analogiczne do tych wyrażonych za pomocą wartości abstrakcyjnych, można je więc zdefiniować tak:

```
(define (ułamek l m) (cons l m))
(define (mianownik q) (car q))
(define (licznik q) (cdr q))
```

7.3.2 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 `ułamek` musi dodatkowo skracać ułamki. Operacje `ułamek`, `mianownik` i `licznik` można zdefiniować tak:

```
(define (ułamek l m)
  (let ((r (nwd l m)))
    (cons (/ l r) (/ m r))))
(define (licznik q) (car q))
(define (mianownik q) (cdr q))
```

Tę reprezentację można dalej ulepszać biorąc pod uwagę znaki licznika i mianownika. Można też, nie obliczać `nwd` przy każdej operacji, ale np. tylko w konstruktorach i selektorach.

7.4 Kiedy bariery abstrakcji są właściwe?

To czy bariery abstrakcji są właściwie poprowadzone możemy sprawdzić wykonując następujący eksperyment myślowy:

1. wymyślamy zestaw zmiad, które chcielibyśmy wprowadzić w programie; zmiany te nie powinny mieć żadnego związku z przyjętą implementacją a jedynie wynikać z rzeczywistych potrzeb użytkownika,

2. badamy, jakie fragmenty programu (na które dzielą program bariery abstrakcji) trzebaby zmodyfikować,

3. im mniej fragmentów w przypadku każdej zmiany należy zmienić, tym lepiej poprowadzone są bariery abstrakcji.

7.5 Podsumowanie zasad z I semestru

- **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 podproblemy, 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,

- **Bariery abstrakcji** — omówione tutaj.

7.6 Struktury listowe i listy

Podobnie jak procedury są równie dobrymi wartościami jak liczby, tak pary są równie dobrymi wartościami jak inne wartości. W szczególności składowymi par mogą być inne pary. Struktury danych, które możemy budować za pomocą par nazywamy *strukturami listowymi* (choć na pierwszy rzut oka wydają się one bliższe drzewom binarnym).

Szczególnym przypadkiem struktury listowej jest lista — struktura reprezentująca ciąg. Ciąg $x_1, x_2, \ldots, x_n$ reprezentujemy jako:

```
(cons x1
  (cons x2
    ⋮
        (cons xn nil) ...))
```

Dla takiej konstrukcji mamy procedurę `list`, z której korzystamy w następujący sposób:

```
(list x1 x2 ...xn)
(define l (list 1 2 3 4))
l
(1 2 3 4)
(car l)
1
(cdr l)
(2 3 4)
```

Mamy też specjalny predykat `null?`, który sprawdza, czy lista jest pusta. Tradycyjnie pierwszy element listy nazywamy jej głową, a listę pozostałych elementów ogonem.

Należy być ostrożnym i nie mylić *wyrażenia* `(list 1 2 3 4)` z listą `(1 2 3 4)`, tak jak ją wypisuje kompilator.

Ponieważ operacje `car` i `cdr` nie są przesadnie poręczne, Scheme udostępnia skróconą formę zapisu ich zlepków, i tak zamiast `(car (cdr l))` możemy napisać `(cadr l)`.

7.6.1 Operacje na listach

Przykładem operacji na listach może być długość listy:

```
(define (length l)
  (if (null? l)
    0
    (+ 1 (length (cdr l)))))
```

Inny przykład to sklejanie list:

```
(define (append l1 l2)
  (if (null? l1)
    l2
    (cons (car l1) (append (cdr l1) l2))))
```

7.6.2 Notacja kropki i ogona

Możemy definiować procedury, które mogą mieć różną liczbę argumentów. Ich definicja ma postać:

```
(define (⟨identyfikator⟩+ . ⟨identyfikator⟩)
  (⟨treść⟩))
```

Wartością parametru po kropce jest lista wartości pozostałych argumentów. Na przykład:

```
(define (p x y . z) (...))
```

Procedura `p` ma przynajmniej dwa argumenty. Pierwszy argument jest równy `x`, drugi `y`, a pozostałe argumenty tworzą listę równą `z`.

Ćwiczenia

Ćwiczenie programistyczne na listy:

- lista n pierwszych liczb naturalnych,
- n-ty element listy,
- rozwinięcie listy list do listy,
- zdublować elementy listy,
- suma dowolnej liczby elementów,
- odwrócenie listy `reverse`,
- napisz program wybierający większość z listy większościowej; uzasadnij jego poprawność.

  Uwaga: to zadanie można rozwiązać probabilistycznie:

  - metoda Las Vegas: wylosuj element i sprawdź czy to większość; policz oczekiwaną złożoność,
  - metoda Monte Carlo: wylosuj k elementów i wybież spośród nich większość, oszacuj prawdopodobieństwo pomyłki w zależności od k. (Sprawdzić czy to prawda?)
- obliczenie n-tego ciągu różnicoweg zadanej listy liczb całkowitych,

8 Zasada dziel i zwyciężaj na przykładzie sortowania

Sformułowanie problemu. Dany jest ciąg n elementów, ze zbioru (nieograniczonej mocy), na którym jest określony porządek liniowy $<$ 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 sortnowania.

Eliminacja algorytmów randomizowanych Jeśli nasz algortym jest randomizacyjny, to ustalamy z góry ciąg losowanych liczb i analizujemy go, jak deterministyczny.

Drzewa decyzyjne W węzłach wewnętrznych mamy warunki dotyczące danych. W liściach umieszczamy wyniki. Obliczenie polega na przejściu od korzenia drzewa do liścia.

Fakt: Jeśli drzewo ma k liści, to ma wysokość nie mniejszą niż $\log_2 k$.

Dowód: Indukcyjnie. Dla $k = 1$ oczywiste. Dla większych k, podzielmy nasze drzewo na dwa poddrzewa. Któreś z nich ma co najmniej $\lceil \frac{k}{2} \rceil$ liści. W związku z tym jest wysokości co najmniej

$$\log_2 \lceil \frac{k}{2} \rceil \geq \log_2 \frac{k}{2} = \log_2 k - 1$$

Czyli dane drzewo jest wysokości co najmniej $\log_2 k$. □

Lemat: $3^n \cdot n! \geq n^n$

Dowód: Indukcyjnie. $3 \geq 1$.

$$3^{n+1} n!(n+1) \geq 3(n+1)n^n \geq e(n+1)n^n \geq \left(1 + \frac{1}{n}\right)^n (n+1)n^n \geq (n+1)^{n+1}$$

□

Twierdzenie 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 decycyjnego, 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

$$h \geq \log_2 \left(\frac{n^n}{3^n}\right) = n \log_2 n - nlog_2 3 = \Omega(n \log n)$$

□

Analiza kilku algorytmów sortowania. Jak realizujemy zasadę dzieli i zwyciężaj.

8.1 Selection sort

Idea algorytmu — redukcja = wybór maksimum. Realizacja:

```
(define (select-max l)
  (define (select m w l)
    (cond
      ((null? l) (cons m w))
      ((>(car l) m) (select (car l) (cons m w) (cdr l)))
      (else (select m (cons (car l) w) (cdr l)))))
  (select (car l) nil (cdr l)))

(define (selection-sort l)
  (define (sort a l)
    (if (null? l) a
      (let ((sm (select-max l)))
        (sort (cons (car sm) a) (cdr sm)))))
  (sort nil l))
```

Analiza kosztu.

8.2 Insertion sort

Idea algorytmu — redukcja = posortuj listę bez jednego elementu i wstaw go do posortowanej listy. Realizacja:

```
(define (insertion-sort l)
  (define (insert x l)
    (define (ins a x l)
      (cond
        ((null? l) (reverse (cons x a)))
        ((>x (car l)) (ins (cons (car l) a) x (cdr l)))
        (else (append (reverse a) (cons x l)))))
    (ins nil x l))
  (define (sort w l)
    (if (null? l) w (sort (insert (car l) w) (cdr l))))
  (sort nil l))
```

Analiza kosztu.

8.3 Sortowanie przez scalanie

Idea algorytmu — podziel listę, posortuj powstałe listy i scal je. Realizacja:

```
(define (merge-sort l)
  (define (split l)
    (define (spl l1 l2 l)
      (cond
        ((null? l) (cons l1 l2))
        ((null? (cdr l)) (cons (cons (car l) l1) l2))
```

```
      (else (spl (cons (car l) l1) (cons (cadr l) l2) (cddr l)))))
    (spl nil nil l))
  (define (merge l1 l2)
    (define (mrg a l1 l2)
      (cond
        ((null? l1) (append (reverse a) l2))
        ((null? l2) (append (reverse a) l1))
        ((>(car l1) (car l2)) (mrg (cons (car l2) a) l1 (cdr l2)))
        (else (mrg (cons (car l1) a) (cdr l1) l2))))
    (mrg nil l1 l2))
  (if (<(length l) 2) l
    (let ((s (split l)))
      (merge (merge-sort (car s)) (merge-sort (cdr s))))))
```

Analiza alożoności.

Pamięć: Rozważmy ile pamięci jest zajętej przy największym zagłębieniu rekurencyjnym. Mamy oryginalną listę (n) oraz jej dwie połówki (n). Jedna z tych połówek może być posortowana ($n/2$), a druga jeszcze nie. Analogicznie, jedna z połówek jest podzielona na dwie ćwiartki ($n/2$), a jedna z ćwiartek może być posortowana ($n/4$), itd. W efekcie uzyskujemy złożoność:

$$M(n) = 2.5n + M(n/2) \leq 2.5 \sum_{k \geq 0} \frac{n}{2^k} = \Theta(n)$$

Czas: Każda faza wymaga liniowego czasu plus dwa razy wywołuje fazy dla swoich połówek.

$$T(1) = 1$$

$$T(n) = 2n + T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor)$$

Czyli dla $n = 2^k$ mamy:

$$T(n) = \sum_{i=0}^{k} 2n = 2(k+1)n = \Theta(n \log n)$$

Możemy pokazać (przez indukcję), że:

$$n\lfloor \log n \rfloor \leq T(n) \leq n \lceil \log n \rceil$$

Ćwiczenia

Listy i funkcje:

- `map`,
- suma listy funkcji,
- złożenie listy funkcji,
- flaga holenderska,

- `fold-left`, `fold-right` i ich zastosowanie do powyższych,
- `filter`,
- Wyznaczanie liczb pierwszych metodą sita Eratostenesa.

Laboratorium

- Zaimplementuj arytmetykę liczb wymiernych ze skracaniem ułamków i unormowaniem znaku.
- Zaimplementuj ulubiony algorytm sortowania (i zachowaj go sobie).

9 Quick-sort

Algorytm quick-sort opiera się na następującym pomyśle: dzielimy elementy ciągu względem wybranego elementu (pierwszego lub losowego) na mniejsze i większe, następnie sortujemy je niezależnie i sklejamy.

```
(define (quick-sort l)
  (define (split x l1 l2 l)
    (cond
      ((null? l) (cons l1 l2))
      ((<= (car l) x) (split x (cons (car l) l1) l2 (cdr l)))
      (else (split x l1 (cons (car l) l2) (cdr l)))))
  (if (<= (length l) 1)
    l
    (let ((s (split (list-ref l (random (length l))) nil nil l)))
      (append (quick-sort (car s)) (quick-sort (cdr s))))))
```

Poprawić: Split powienien dzielić na trzy części, $<$, $=$ i $>$. Dotyczy to też rachunków poniżej.

Pesymistyczna złożoność tego algorytmu nie jest najlepsza, gdyż nie jest on odporny na wybór elementów w porządku rosnącym. W takim przypadku:

$$M(0) = 1$$

$$M(n) = \Theta(n) + M(n-1) = \Theta(n^2)$$

$$T(0) = 1$$

$$T(n) = 2n + T(n-1) = \Theta(n^2)$$

Zbadajmy jednak jak sprawa wygląda średnio.

Twierdzenie: Algorytm Quick-sort, dla losowej permutacji zbioru $\{1, \ldots, n\}$ ma oczekiwaną złożoność czasową $\Theta(n \log n)$.

Dowód: Możemy się skupić na liczbie wykonywanych porównań, gdyż jest ona taka sama jak złożoność czasowa algorytmu. Oznaczmy przez C_n — zmienną losową oznaczającą liczbę porównań przy sortowaniu n elementów.

W trakcie sortowania rozdzielamy listę na dwie listy. Pokażemy, że każda z tych podlist też jest losową permutacją.

Lemat: Niech wybrany element sortowanej listy jest równy s. Listy, na które jest rozdzielana dana lista są losowymi permutacjami zbiorów $\{1, \ldots, s\}$ i $\{s+1, \ldots, n\}$.

Dowód lematu: Na wejściu możemy mieć $n!$ możliwych list. Wystarczy więc pokazać, że każda para wynikowych permutacji odpowiada dokładnie:

$$\frac{n!}{s!(n-s)!} = \binom{n}{s}$$

różnych permutacji na wejściu. Zauważmy, że dla ustalonych list wynikowych, podział elementów w liście wejściowej na te, które trafiły do pierwszej i drugiej listy jednoznacznie wyznaczają listę wejściową. Takich rozdziałów jest jednak dokładnie

$$\binom{n}{s}$$

□

Oznaczmy też przez:

- $c_{n,k,s}$ — prawdopodobieństwo wykonania k porównań pod warunkiem, że wybrany element jest równy s,
- $c_{n,k}$ — prawdopodobieństwo wykonania dokładnie k porównań,
- $C_n(z) = \sum_{k\geq 0} c_{n,k}z^k$ — funkcję tworzącą liczby porównań.

Koszt fazy algorytmu wynosi $n+1$, a wielkość wywoływanych faz wynosi odpowiednio s i $n-s$, czyli:

$$\sum_{k\geq 0} c_{n,k,s}z^k = z^{n+1}C_s(z)C_{n-s}(z)$$

Ponieważ różne s są równoprawdopodobne, więc:

$$c_{n,k} = \sum_{s=1}^{n} \frac{1}{n} c_{n,k,s}$$

Czyli

$$C_n(z) = \sum_{k\geq 0} c_{n,k}z^k = \frac{\sum_{s=1}^{n} z^{n+1}C_s(z)C_{n-s}(z)}{n}$$

Stąd mamy zależność rekurencyjną:

$$C_0(z) = C_1(z) = 1$$

$$C_n(z) = \frac{1}{n} z^{n+1} \sum_{s=1}^{n} C_s(z)C_{n-s}(z)$$

Nas interesuje tylko $C'_n(1)$, którą będziemy oznaczać w skrócie przez C'_n. Ponieważ $C_n(1) = 1$, mamy:

$$C'_n = \frac{1}{n}(n+1)1^n \sum_{s=1}^{n} C_s(1)C_{n-s}(1) + \frac{1}{n}1^{n+1} \sum_{s=1}^{n} (C'_s C_{n-s}(1) + C_s(1)C'_{n-s}) =$$

$$(n+1) + \frac{1}{n}\sum_{s=1}^{n} (C'_s + C'_{n-s}) = (n+1) + \frac{1}{n}\sum_{s=1}^{n} (C'_s + C'_{s-1}) = (n+1) + \frac{2}{n}\sum_{s=1}^{n-1} C'_s + \frac{1}{n}C'_n$$

Stąd:

$$nC'_n = n(n+1) + 2\sum_{s=1}^{n-1} C'_s + C'_n$$

$$(n-1)C'_n = n(n+1) + 2\sum_{s=1}^{n-1} C'_s$$

stąd zaś:

$$(n-1)C'_n - (n-2)C'_{n-1} = n(n+1) - n(n-1) + 2\sum_{s=1}^{n-1} C'_s - 2\sum_{s=1}^{n-2} C'_s = 2n + 2C'_{n-1}$$

tak więc:

$$(n-1)C'_n = nC'_{n-1} + 2n$$

czyli:

$$\frac{C'_n}{n} = \frac{C'_{n-1}}{n-1} + \frac{2}{n-1}$$

Ponieważ $C'_0 = C'_1 = 0$, więc:

$$\frac{C'_n}{n} = \frac{2}{n-1} + \frac{2}{n-2} + \cdots + \frac{2}{1} = 2 \cdot H_{n-1}$$

Czyli:

$$C'_n = 2nH_{n-1}$$

Ponieważ:

$$H_n = \ln n + \gamma + \Theta(n^{-1})$$

gdzie γ jest stałą Eulera, więc:

$$E(C_n) = C'_n = 2n\ln(n-1) + \Theta(n) = \Theta(n \log n)$$

□

Ćwiczenia

- Napisz program wybierający max i min z listy i dokonujący jak najmniejszej liczby porównań,
- Zadanie o katastrofach lotniczych: Dana jest lista liczb całkowitych `(list` $x_1 \dots x_n$`)`. Dla każdego i trzeba policzyć największe takie k_i, że $x_i = \max(x_{i-k_i+1}, \dots x_i)$. Przyjmujemy, że $x_0 = \infty$. (Wymaga użycia stosu.)
- Dana jest lista liczb całkowitych `(list` $x_1 \dots x_n$`)`. Dla każdego i trzeba policzyć największe takie otoczenie x_i, czyli podciąg $x_{i-k_i}, \dots x_{i+k_i}$, że $x_i = \max(x_{i-k_i}, \dots x_{i+k_i})$. (Przez sprowadzenie do poprzedniego.)
- Maksymalne plateau w liście.
- Zbiory można zaimplementować jako listy uporządkowane. Zaimplementuj operacje na zbiorach. Znajdź procedurę wyższego rzędu, która ujmuje wspólny schemat „scalania” uporządkowanych list i zaimplementuj operacje na zbiorach za jej pomocą.
- Udowodnij, że drzewo binarne o n węzłach ma wysokość $\lfloor \log_2 n \rfloor$,

10 Heap sort i kolejki

10.1 Kolejka priorytetowa

Posłużmy się ponownie pobożnymi życzeniami. Załóżmy, że mamy dostępną strukturę danych zwaną *kolejką priorytetową*. W kolejce takiej przechowujemy elementy ze zbioru z określonym porządkiem liniowym. Wartości (abstrakcyjne) tego typu możemy sobie przedstawić jako zbiory (lub lepiej multizbiory) wartości elementów. Dostępne są następujące operacje na takiej kolejce:

- `empty` — pusta kolejka,
- `put` — włóż element do kolejki,
- `getmax` — wyjmij z kolejki największy element,
- `empty?` — sprawdza czy kolejka jest pusta.

10.1.1 Sortowanie

Mając dostępną taką kolejkę możemy zrealizować np. sortowanie:

```
(define (sort l)
  (define (putall q l)
    (if (null? l)
      q
      (putall (put q (car l)) (cdr l))
    )
  )
  (define (getall l q)
    (if (empty? q)
      l
      (let ((gm (getmax q)))
        (getall (cons (car gm) l) (cdr gm))
      )
    )
  )
  (getall nil (putall empty l))
)
```

10.1.2 Realizacja listowa — selection sort

Możliwych jest wiele realizacji kolejek priorytetowych. Zacznijmy od nieefektywnej, ale prostej realizacji listowej. Elementy kolejki pamiętamy na liście, w dowolnej kolejności. Funkcja abstrakcji przyporządkowuje liście multizbiór jej elementów.

```
(define empty nil)
(define empty? null?)
(define (put q x) (cons x q))
(define (getmax q)
```

```
  (define (get a r l)
    (if (null? l)
      (cons a r)
      (if (>a (car l))
        (get a (cons (car l) r) (cdr l))
        (get (car l) (cons a r) (cdr l))
      )
    )
  )
  (if (null? q)
    (error Ęmpty queue")
    (get (car q) nil (cdr q))
  )
)
```

Algorytm sortowania oparty na takiej realizacji kolejki, to sortowanie przez wybieranie.

10.1.3 Realizacja listowa — insertion sort

Możemy też zastosować inną implementację listową kolejki priorytetowej. Konkretne wartości kolejek, to dowolne listy o elementach uporządkowanych nierosnąco. Funkcja abstrakcji, podobnie jak poprzednio, przyporządkowuje kolejce multizbiór jej elementów. Jednak tym razem jest to fukcja 1–1.

```
(define empty nil)
(define empty? null?)
(define (put q x)
  (define (iter l x r)
    (if (null? r)
      (reverse (cons x l))
      (if (>(car r) x)
        (iter (cons (car r) l) x (cdr r))
        (append (reverse (cons x l)) r)
      )
    )
  )
  (iter nil x q)
)
(define (getmax q)
  (if (null? q)
    (error Ęmpty queue")
    q
  )
)
```

Algorytm sortowania oparty na takiej realizacji kolejki, to sortowanie przez wstawianie.

10.1.4 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 węzła spełniony jest następujący warunek: wartość przechowywana w węźle jest większ lub równa wszystkim wartościom przechowywanym w poddrzewie, którego ten węzeł jest korzeniem. Węzeł drzewa to lista czteroelementowa: wartość, liczba elementów, lewe poddrzewo, prawe poddrzewo.

Sposób realizacji struktury danych reprezentujemy za pomocą zestawu pomocniczych selektorów/konstruktorów:

```
(define root car)
(define (size h) (if (null? h) 0 (cadr h)))
(define left caddr)
(define right cadddr)
(define (heap v l r) (list v (+ (size l) (size r) 1) l r))
```

Procedury te stanowią barierę abstrakcji oddzielającą pojęcie stogu od jego implementacji za pomocą struktur listowych.

Kolejna warstwa to implementacja operacji na kolejce priorytetowej za pomocą stogu.

```
(define empty nil)
(define empty? null?)
(define (put h x)
  ;; Rotacje w górę przy wstawianiu do stogu
  (define (rotate h)
    ;; Podniesienie korzenia lewego poddrzewa
    (define (lift-left h)
      (let
          ((l (left h)))
        (heap
          (root l)
          (heap (root h) (left l) (right l))
          (right h))))
    ;; Podniesienie korzenia prawego poddrzewa
    (define (lift-right h)
      (let
          ((r (right h)))
        (heap
          (root r)
          (left h)
          (heap (root h) (left r) (right r)))))
    (cond
      ((empty? h) h) ; Pusty stóg --bez rotacji
      ((and (empty? (left h)) (empty? (right h))) h) ; Jeden węzeł --bez
rotacji
      ((empty? (right h)) ; Tylko lewe poddrzewo
        (if (<(root h) (root (left h))) (lift-left h) h))
      ((empty? (left h))  ; Tylko prawe poddrzewo
```

```
          (if (<(root h) (root (right h))) (lift-right h) h))
        (else  ; Oba poddrzewa są niepuste
          (cond
            ((and (>= (root h) (root (left h)))
                  (>= (root h) (root (right h))))
              h)
            ((>= (root (left h)) (root (right h))) (lift-left h))
            (else (lift-right h)))
        ))
    )
    (if (empty? h)
        (heap x empty empty)
        (if (<= (size (left h)) (size (right h)))
            (rotate (heap (root h) (put (left h) x) (right h)))
            (rotate (heap (root h) (left h) (put (right h) x)))
        )
    )
)
```

Załóżmy, że dokonujemy n operacji `put` i n operacji `getmax`.

Fakt Wysokość drzewa stogu nie przekracza $\lfloor \log_2 n \rfloor$.

Dowód: Zauważmy, że gdyby były tylko operacje wstawiania (lub najpierw wstawialibyśmy elementy, a potem je wyjmowali), to drzewo nie miałoby przez to większej wysokości. Jeśli mamy same wstawienia, to zauważmy, że dla każdego węzła zachodzi następujący warunek: liczba wierzchołków w lewym poddrzewie i liczba wierzchołków w prawym poddrzewie różnią się o co najwyżej 1. Stąd wynika, że jeżeli mamy same wstawienia, to mamy drzewo zrównoważone. Ponieważ zrównoważone drzewo binarne o n wierzchołkach ma wysokość $\lfloor \log_2 n \rfloor$ uzyskujemy tezę.

Zauważmy, że koszt (czasowy i pamięciowy) 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 $\Theta(n)$.

10.2 Koszt zamortyzowany

Przy okazji stogu zastanówmy się nad takim problemem: jeśli mamy ciąg wstawień i usunięć elementów ze stogu, przy czym (zwykle) stóg zawiera istotnie mniej elementów niż n, to czy potrafimy lepiej oszacować koszt operacji wstawiania i usuwania? Jak zauważyliśmy wcześniej, koszt wstawiania jest proporcjonalny do logarytmu z liczby węzłów w stogu, a koszt usuwania jest co najwyżej proporcjonalny do wysokości stogu. Zauważmy, że wysokość stogu może być istotnie większa niż $\log_2$ z liczby aktualnie przechowywanych w nim elementów.

Okazuje się, że możemy „sumarycznie” przyjąć, że:

- koszt operacji wstawiania jest $O(\log n)$, gdzie n jest aktualnym rozmiarem stogu,

- 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 torchę jej pozostawiamy zmagazynowanej w stogu. A zatem, łączny czas obliczeń jest *co najwyżej* taki, jak ilość zużytej energii.

Bardziej formalnie, postępujemy następująco:

- określamy *funkcję amortyzacji* 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 amortyzacji,
- po zakończeniu obliczeń możemy od kosztu odjąć różnicę między wartościa funkcji amortyzacji dla końcowej i początkowej wartości struktury danych.

10.3 Kolejka FIFO — przykład kosztu zamortyzowanego

Kolejka FIFO to taka kolejka, z której wyjmujemy elementy w takiej kolejności, w jakiej były wstawiane. 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.

```
(define (fifo in out size)
  (define (balance in out)
    (cond
      ((null? in) (cons in out))
      ((null? out) (cons nil (reverse in)))
      (else (cons in out))))
```

```
  (let
      ((inout (balance in out)))
    (let
        ((in1 (car inout)) (out1 (cdr inout)))
      (let
          ((first (if (null? out1) nil (car out1))))
        (list in1 out1 size first)))))
(define empty (fifo nil nil 0))
(define inqueue car)
(define outqueue cadr)
(define size caddr)
(define cache cadddr)
(define (empty? q) (= (size q) 0))
(define (first q)
  (if (>(size q) 0)
      (cache q)
      (error Ęmpty queue!")))
(define (put q x)
  (fifo (cons x (inqueue q)) (outqueue q) (+ (size q) 1)))
(define (remove q)
  (if (empty? q)
      (error Ęmpty queue!")
      (fifo (inqueue q) (cdr (outqueue q)) (-(size q) 1))))
```

Zanalizujmy koszt zamortyzowany operacji. Funkcja amortyzacji to `(length (inqueue q))`. Procedury `empty`, `empty?`, `size` i `first` mają koszt stały i nie zmieniają funkcji amortyzacji. Dla pustej kolejki funkcja amortyzacji jest równa 0. Procedura `put` ma koszt stały i powoduje zwiększenie funkcji amortyzacji o 1, co daje stały koszt zamortyzowany. Procedura `get` może mieć koszt liniowy, ze względu na procedurę lokalną `balance`. Jednak ta ostatnia wykonuje tyle kroków, o ile zmniejsza funkcję amortyzacji. Tak więc koszt zamortyzowany procedury `get` jest stały.

Ćwiczenia

- gdzie w stogu znajdują się elementy minimalne?
- sformułuj taki warunek określający, czy element ma być wstawiony do lewego, czy prawego poddrzewa, aby kolejne elementy były wstawiane na kolejnych poziomach od lewej do prawej,
- napisz funkcję, która przekształci zadane drzewo w stóg, zachowując jego kształt,
- napisz funkcję, która „wyważy” zadany stóg, to znaczy tak poprzestawia w nim elementy, aby miał minimalną wysokość; jaka jest jej złożoność?
- Rozszerz implementację kolejek FIFO tak, aby obsługiwały operacje LIFO+FIFO w koszcie zamortyzowanym stałym.
- Ćw 2.29: mobile.

10.4 Abstrakcja rekurencji

Wszystkie funkcje (z ćwiczeń) operujące na listach mają z grubsza taki sam schemat. Czy można napisać funkcję będącą kwintesencją tego schematu?

```
let rec fold a f = function [] ->a | (h::t) ->fold (f (a,h)) f t;;
let length = fold 0 (function (a,_) ->a + 1);;
let sum = fold 0 (function (a,b) ->a + b);;
```

Czy istnieje kwintesencja wszelkiej rekurencji? A co to jest rekurencja? Jak przerobić definicję rekurencyjną na nierekurencyjną — wprowadzenie dodatkowego parametru funkcyjnego. Przykład: silnia. Z rekurencją:

```
let rec fac n =
  if n <= 1 then 1 else fac (n -1);;
```

Bez rekurencji:

```
let factorial fac n =
  if n <= 1 then 1 else fac (n -1);;
```

To jest typu `(int` $\to$ `int)` $\to$ `(int` $\to$ `int)`. Teraz trzeba jakoś chytrze podstawić funkcję wynikową jako argument. Inaczej mówiąc szukamy funkcji, która jest punktem stałym `factorial`. Sprawdzenie: zakładając, że `f` jest punktem stałym `factorial` policz `f 3`.

Operator punktu stałego — pierwsze podejście.

```
let rec y f = f (y f);;
let f = y factorial ;;
```

To się niestety zapętli.

Uleniwianie Dowolną wartość możemy uleniwić, zamieniając ją na funkcję wymagającą nic nie znaczącego argumentu.

```
let delay x = function () ->x;;
let force x = x ();;
let dużo = delay 1234;;
let cosik = delay 12;;
force dużo + force cosik;;
```

Aby uniknąć zapętlenia, musimy uleniwić nasz dodatkowy argument i wyliczać go tylko na rządanie. Operator punktu stałego.

```
let factorial fac n =
  if n <= 1 then 1 else n * (force fac (n -1));;
let rec y f = f (function () ->y f);;
let f = y factorial ;;
```

Rozpisać wyprowadzenie `f 3;;`.

11 Funkcje jako wartości reprezentujące dane

11.1 Wartości logiczne

Aby mówić o wartościach logicznych potrzebuję:

- prawdę i fałsz,
- `if` ... `then` ... `else` ...,
- koniunkcję, alternatywę i negację.

Wartość logiczną reprezentują jako funkcję (typu $\alpha \to \alpha \to alpha$), która w przypadku prawdy wybiera pierwszy, a w przypadku fałszu drugi argument.

```
let prawda x y = x;;
let fałsz x y = y;;
let jeśli w k a = force (w k a);;
jesli prawda (delay 1) (delay 2);; = 1
let i x y = x y fałsz;;
let lub x y = x prawda y;;
jeśli (lub fałsz prawda) (delay 1) (delay 2);; = 1
```

11.2 Produkty kartezjańskie

. Aby mowić o produkcie muszę mieć:

- konstruktor `pr:` $\alpha \to \beta \to$ `produkt`,
- rzuty: `p1: produkt` $\to \alpha$ i `p2: produkt` $\to \beta$.

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 = pr 1 ‘‘ala’’;;
p1 p;; = 1
p2 p;; = ‘‘ala’’
```

11.3 Listy

Listy nieskończone można reprezentować jako funkcje z `int`. Potrzebujemy:

- listę stałą,
- dołączenie elementu na początek listy,
- pierwszy element listy,
- ogon listy.

```
let stała x n = x;;
let sklej h t n = if n = 0 then h else t (n-1);;
let pierwszy l = l 0;;
let ogon l n = l (n + 1);;
```

11.4 Liczby naturalne Churcha

Pozostały nam jeszcze liczby naturalne. Utożsamiamy liczbę naturalną z liczbą iteracji zadanej funkcji.

```
let id x = x;;
let złóż f g = function x ->f (g x);;
let zero f = id;;
let jeden f = f;;
```

Dodawanie i mnożenie:

```
let inc n f = złóż (n f) f;; let plus m n f = złóż (m f) (n f);;
let razy = złóż;;
```

OK, ale jak z tych liczb korzystać?

```
let test_zera x = x (function _ ->fałsz) prawda;;
jesli (test_zera zero) (delay 1) (delay 2);; = 1
let dec n f x =
  let (y,_) = n (function (a,b) ->(b,f b)) (x,x)
  in y;;
val dec: (('a*'b ->'b*'c) ->'d*'d ->'e*'f) ->
('b ->'c) ->'d ->'e
```

Do celów testowych można używać:

```
let ile n = n (function x ->x + 1) 0;;
let liczba =
  let rec licz a n f =
    if n = 0 then a else licz (złóż a f) (n -1) f
  in
    licz id;;
ile (liczba 1000000);; = 1 000 000
```

Laboratorium

- zadanie o Origami – lista prostych zorientowanych wyznaczająca ciąg złożeń papieru, dla danego punktu policzyć ile warstw papieru się przebije.

  Jest to bardzo ładne zadanie. Należy tu wykorzystać funkcje jako strukturę danych reprezentującą stan kartki.

- Arytmetyka przedziałów.

Zadanie domowe

Dana jest drzewo typu:

```
type choinka =
  Koniuszek |
  Galaz of choinka |
  Rozgalezienie of choinka * choinka;;
```

opisujące kształt choinki. W każdym wierzchołku tego drzewa należy zawiesić bombkę. 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 złota wysokiej próby, 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 funkcję `bombki: choinka -> int`, która dla danej choinki obliczy na ile sposobów można powiesić na niej bombki.

12 Kody Huffmana

- Kody stałej długości (np. ASCII) i 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:
BACADAEAFABBAAAGAH
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 20% pamięci.

- Problem zdefiniowania optymalnego kodu prefiksowego przy znanych (względnych) częstościach występowania znaków.
- Reprezentacja kodowania w postaci regularnego drzewa binarnego.
- Algorytm Huffmana — opis.
- Prześledzić opis na przykładzie.
- 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 pregiksowy dla C, w którym kody dla x i y mają tę samą długość i różnią się tylko na ostatnim bicie.

Dowód: Rozważmy pewne optymalne kodowanie. Niech b i c będą dwoma sąsiednimi liśćmi o maksymalnej głębokości. 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 x i y. Zamieniamy b z x i c z y. Tak powstałe drzewo daje nie gorsze kodowanie, a więc również optymalne.

Lemat [CLR] Niech T będzie regularnym drzewem binarnym odpowiadającym optymalnemu kodowaniu alfabetu C, gdzie dla każdego $c \in C$ częstość występowania c wynosi $f(c)$. Niech x i y będą znakami (liśćmi), a z będzie ich ojcem. Wówczas, jeżeli zastąpimy x i y znakiem z o częstości występowania $f(z) = f(x) + f(y)$, i T' będzie drzewem powstałym przez usunięcie z T liści x i y, to będzie to optymalne drzewo kodowania alfabetu $C' = C - \{c\}$.

Dowód: Gdyby nie było to kodowanie optymalne, to możnaby je poprawić. Wówczas, w tak poprawionym drzewie, możemy wierzchołkowi z dodać dwóch synów x i y. Tak uzyskane drzewo określałoby lepsze kodowanie niż początkowe, a to było optymalne — stąd sprzeczność.

Z tych dwóch lematów wynika natychmiast poprawność algorytmu.

- Schemat programowania zachłannego:
  - szukamy pewnego rozwiązania optymalnego, robimy to za pomocą ciągu wyborów („lokalnych") kolejnych elementów rozwiązania,
  - własność wyboru zachłannego — ciąg wyborów optymalnych lokalnie prowadzi nas do rozwiązania optymalnego globalnie,
  - 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,
  - własność optymalnej podstruktury — optymalne rozwiązanie jest funkcją optymalnych rozwiązań podproblemów,
  - własność ta upraszcza dowód własności wyboru zachłannego — wystarczy pokazać, że pewne rozwiązanie optymalne składa się z dokonanego wyboru i optymalnego wyboru podproblemu.

Laboratorium

Opowiedzieć o modułach.

Tam gdzie dokonujemy pomiarów wielkości fizycznych, wyniki są obarczone pewnym błędem, np. 5m +/- 10%. Zaimplementuj pakiet operacji arytmetycznych na takich przybliżonych wartościach zawierający:

- konstruktory:
  - `wartość_dokładność x p` = `x` $\pm$ `p%`,
  - `wartość_od_do x y` = `(x+y)/2` $\pm$ `(y-x)/2` (dla `x` $\leq$ `y`),
  - `wartość_dokładna x` = dokładnie `x`,
- selektory:
  - `min_wartości x` = najmniejsza możliwa wartość `x`,
  - `max_wartości x` = największa możliwa wartość `x`,
  - `śr_wartości x` = średnia (arytmetyczna) wartość `x`,
- modyfikatory:
  - plus,
  - minus,
  - razy,
  - podzielić.

Konkretna reprezentacja wartości powinna być ukryta za (własnymi, wewnętrznymi) konstruktorami i selektorami. Selektory, w przypadku, gdy wynik nie jest liczbą rzeczywistą, powinny dawać wyniki, odpowiednio, `inf`, `-inf`, `nan`.

Ćwiczenia

- ciągły problem plecakowy,
- [CLR] p. 17.1, problem wyboru zajęć (historyjka o kinomanie i jak największej liczbie filmów do obejrzenia),
- [CLR] ćw 17.2-4, problem tankowania na stacjach benzynowych,

13 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`, `huff_remove`.

13.1 Implementacja drzew Huffmana

```
type 'a huff_tree =
  Char of ('a * int) |
  Node of ('a huff_tree * 'a huff_tree * int);;

let frequency (t : 'a huff_tree) =
  match t with
    Char (_, f) -> f |
    Node (_, _, f)   -> f;;

let merge t1 t2 = Node (t1, t2, frequency t1 + frequency t2);;

huffman ['C',1; 'D',1; 'E',1; 'F',1; 'G',1; 'H',1; 'B',3; 'A',8];;

Node
 (Char ('A', 8),
  Node
   (Node
     (Node (Char ('C', 1), Char ('D', 1), 2),
      Node (Char ('E', 1), Char ('F', 1), 2), 4),
    Node (Node (Char ('G', 1), Char ('H', 1), 2), Char ('B', 3), 5), 9),
  17)
```

Należy zwrócić uwagę, że deklaracja datatypu wprowadza konstruktory i selektory spełniające określone własności.

13.2 Implementacja kolekcji drzew Huffmana

```
(* Kolekcja drzew Huffmana *)
type 'a huff_col = 'a huff_tree fifo * 'a huff_tree fifo;;

(* Tworzy kolekcje lisci Huffmana na podstawie listy czestosci
   wystepowania uporzadkowanej niemalejaco *)

let make_huff_col (l: ('a * float) list) =
  let rec queue q l =
    match l with
      []   -> q |
      h::t -> queue (put_last q (Char h)) t
  in
    ((queue empty l, empty): 'a huff_col);;

let huff_size ((q1, q2): 'a huff_col) =
  fifo_size q1 + fifo_size q2;;

let huff_put ((q1, q2): 'a huff_col) t =
  ((q1, put_last q2 t): 'a huff_col);;

let huff_first ((q1, q2): 'a huff_col) =
  if q1 = empty then
    first q2
  else if q2 = empty 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 then
    (empty, remove q2)
  else if q2 = empty then
    (remove q1, empty)
  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);;
```

- Implementacja kolejek FIFO. Pokażemy tu bardziej ogólną strukturę — kolejki dwukierunkowe.

```
(* Typ = przód, tył, pierwszy, ostatni *)
type 'a kolejka = 'a list * 'a list * int;;

(* Pusta kolejka *)
let empty : 'a kolejka = ([], [], 0);;

(* Rozmiar *)
let size ((_, _, n): 'a kolejka) = n;;

(* Wyrównanie listy, jeśli jedna z połówek jest pusta, a druga
   zawiera więcej niż jeden element. *)
let balance (q: 'a kolejka) =
  (* Podział listy na połowy h1 i h2, length h1 <= lenght h2 *)
  let halve l =
    let rec head l n a =
      if n = 0 then
        (List.reverse a, l)
      else
        match l with
          []   -> failwith "Zbyt krótka lista" |
          h::t -> head t (n-1) (h::a)
    in
      head l (List.length l / 2) []
  in
    match q with
      ([], [], _)    -> q |
      ([_], [], _)   -> q |
      ([], [_], _)   -> q |
      (front, [], n) ->
        let (h1, h2) = halve front
        in (h1, List.reverse h2, n) |
      ([], back, n)  ->
        let (h1, h2) = halve back
        in (List.reverse h2, h1, n) |
      _           -> q ;;

(* Wstawienie na początek *)
let put_first ((front, back, n): 'a kolejka) x =
  ((balance (x::front, back, n+1)): 'a kolejka);;
```

```
(* Wstawienie na koniec *)
let put_last ((front, back, n): 'a kolejka) x =
  ((balance (front, x::back, n+1)): 'a kolejka);;

(* Początek *)
let first (q: 'a kolejka) =
  match q with
    ([], [x], _) -> x |
    ([], _, _)   -> failwith "Pusta kolejka" |
    (x::_, _, _) -> x;;

(* Koniec *)
let last (q: 'a kolejka) =
  match q with
    ([x], [], _) -> x |
    (_, [], _)   -> failwith "Pusta kolejka" |
    (_, x::_, _) -> x;;

(* Wyjęcie początku *)
let remove_first (q: 'a kolejka) =
  match q with
    (_::front, back, n) -> balance (front, back, n-1) |
    _ -> empty;;

(* Wyjęcie końca *)
let remove_last (q: 'a kolejka) =
  match q with
    (front, _::back, n) -> balance (front, back, n-1) |
    _ -> empty;;

(* Konstrukcja kolejki na podstawie listy *)
let make_queue l =
  let rec make l a =
    match l with
      []   -> a |
      h::t -> make t (put_last a h)
  in
    make l empty;;
```

Laboratorium

Praca własna nad zadanym zadaniem.

Ćwiczenia

Zadania rozwiązuje się (mniej lub bardziej) zachłannie. Dla każdego z nich, oprócz programu należy uzasadnić poprawność rozwiązania zachłannego.

- 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.

- [IV OI] 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.

14 Algorytmy zachłanne c.d. — problem kajakowy

Problem: Jak usadzić n osób o podanych wagach w minimalnej liczbie kajaków; każda osoba sama mieści się w kajaku, ale dwie osoby mogą już przekroczyć określoną, stałą, wyporność kajaka; wagi osób są dane w postaci listy uporządkowanej niemalejąco; w kajaku może płynąć tylko jedna lub dwie osoby.

Problem ten możemy rozwiązać zachłannie i to na kilka sposobów. W rozwiązaniach tych będziemy korzystać z dwustronnej kolejki.

1. Każdemu grubasowi znajdujemy jak najgrubszego chudzielca, który może płynąć razem z nim. Zauważmy przy tym, że im chudszy grubas, tym grubszy może być chudzielec.

   ```
   type kajak = Jedna of int | Dwie of (int * int);;

   let kajaki l wypornosc =
     let rec dobierz g ch =
       if g = empty then
         if  ch = empty then
           (empty, empty)
         else
           (put_last empty (first ch), remove_first ch)
       else
         if size g = 1 then
           (g, ch)
         else
           if first g + last g <= wypornosc then
             dobierz (remove_last g) (put_first ch (last g))
           else
             (g, ch)
     and sadzaj grubi chudzi wynik =
       let (g, ch) = dobierz grubi chudzi
       in
         if g = empty then
           wynik
         else if ch <> empty then
           sadzaj (remove_first g) (remove_first ch)
                  (Dwie (first g, first ch)::wynik)
         else
           sadzaj (remove_first g) ch (Jedna (first g)::wynik)
     in
       sadzaj (make_queue (List.reverse l)) empty [];;
   ```

   Uzasadnienie poprawności.

2. Dla największego grubasa możemy też starać się dobrać najmniejszego chudzielca:

   ```
   (* Rozwiązanie III
      Jeśli się da, to łączymy skrajne. *)
   ```

```
let kajaki l wypornosc =
  let rec sadzaj q wynik =
    if size q < 2 then
      if q = empty then
        wynik
      else
        Jedna (first q) :: wynik
    else
      if first q + last q <= wypornosc then
        sadzaj (remove_first (remove_last q))
               (Dwie (first q, last q) :: wynik)
      else
        sadzaj (remove_last q) (Jedna (last q) :: wynik)
  in
    sadzaj (make_queue l) [];;
```

3. Dla każdego chudzielca dobieramy jak największego grubasa, który może z nim siedzieć:

```
let kajaki l wypornosc =
  let rec dobierz ch g=
    if ch = empty then
      (ch, g)
    else
      if size ch = 1 then
        (empty, put_first g (first ch))
      else
        if first ch + last ch > wypornosc then
          dobierz (remove_last ch) (put_first g (last ch))
        else
          (ch, g)
  and sadzaj chudzi grubi wynik =
    let (ch, g) = dobierz chudzi grubi
    in
      if ch = empty then
        if g = empty then
          wynik
        else
          sadzaj empty (remove_first g) (Jedna (first g) :: wynik)
      else
        sadzaj (remove_first (remove_last ch)) g
               (Dwie (first ch, last ch) :: wynik)
  in
    sadzaj (make_queue l) empty [];;
```

Uzasadnienie poprawności.

4. Możemy też wsadzać do kajaka parę dwóch kolejnych, jak najgróbszych. Nie potrzeba do tego kolejek.

```
let kajaki4 l wypornosc =
  let rec dobierz ch g=
    match (ch, g) with
      (ch, [])   -> (ch, []) |
      ([], x::t) -> dobierz [x]  t |
      (x1::t1, x2::t2) ->
        if x1 + x2 <= wypornosc then
          dobierz (x2::(x1::t1)) t2
        else
          (ch, g)
  and sadzaj chudzi grubi wynik =
    let (ch, g) = dobierz chudzi grubi
    in
      match (ch, g) with
        ([], [])     -> wynik |
        ([], x::t)   -> sadzaj [] t (Jedna x :: wynik) |
        ([x], _)     -> sadzaj [] g (Jedna x :: wynik) |
        (x::y::t, _) -> sadzaj t g (Dwie (x, y) :: wynik)
  in
    sadzaj [] l [];;
```

Uzasadnienie poprawności.

Ćwiczenia

- [V OI - uproszczone] Dana jest liczba $n \geq 0$ oraz $0 \leq s \leq \frac{n(n+1)}{2}$. Podaj podzbiór zbioru $\{1, \ldots, n\}$, którego suma elementów jest równa s.
- [II OI(?)] Podziel dane słowo (listę znaków) na maksymalną liczbę palindromów parzystej długości. Złożoność $O(n^2)$.
- [II OI] Jak obejść wszystkie wierzchołki drzewa (ogólnego) tak, aby odległość między kolejnymi wierzchołkami nie przekraczała 3.

15 Programowanie dynamiczne — najdłuższy wspólny podciąg

15.1 Problem i algorytm

- Problem najdłuższego wspólnego podciągu.
- Uogólnienie problemu na prefiksy danych podciągów.
- Własność podproblemu, zależności między rozwiązaniami podproblemów.
  - Jeśli $x_i = y_j$, to długość $nwp(i, j) = nwp(i-1, j-1) + 1$.
  - Jeśli ostatni wyraz nwp jest różńy 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.
  - Stąd, jeśli $x_i \neq y_j$, to $nwp(i, j) = \max(nwp(i-1, j), nwp(i, j-1))$.
  - Warunki brzegowe — zero.
- Algorytm — tablica rozwiązań dla podproblemów i jej wypełnianie.

```
let dlugosc u v w x y =
  if x = y then
    v + 1
  else
    max u w;;

let nwp x y =
  let
    a = zbuduj_tablice x y
  in
    odtworz_podciag a x y;;
```

15.2 Implementacja nieefektywna — mapy

- Koncepcja mapy/słownika/funkcji częściowej.
  - `exception Undefined;;`
  - `apply m x`
  - `empty_map`
  - `dom m x`
  - `update f a v`
- Implementacja w oparciu o mapy.
  - Główna procedura pozostaje bez zmian:

```
let nwp x y =
  let
    a = zbuduj_tablice x y
  in
    odtworz_podciag a x y;;
```

– Budowa tablicy:

```
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 rec dodawaj tablica lista i =
      match lista with
        []     -> tablica |
        (x::t) ->
          let u = apply tablica (j, i-1)
          and v = apply tablica (j-1, i-1)
          and w = apply tablica (j-1, i)
          in
            let
              tab = update tablica (j, i) (dlugosc u v w x y)
            in
              dodawaj tab t (i+1)
    in
      dodawaj (update tablica (j, 0) 0) ciag_x 1
  in
    (* Pierwszy wiersz tablicy -- same zera *)
    let rec pierwszy akumulator n =
      if n < 0 then
        akumulator
      else
        pierwszy (update akumulator (0, n) 0) (n-1)

    (* Rekurencyjne budowanie tablicy.
       yl= pozostale elementy ciągu y, j = nr wiersza.
    *)
    and buduj akumulator yl j =
      match yl with
        []     -> akumulator |
        (y::t) -> buduj (dodaj_wiersz akumulator y j) t (j+1)
    in
      buduj (pierwszy empty_map (List.length ciag_x)) ciag_y 1;;
```

– Rekonstrukcja podciągu:

```
let odtworz_podciag tablica ciag_x ciag_y =
  let rec odtworz akumulator 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) tx ty (i-1) (j-1)
              else
                let u = apply tablica (j, i-1)
                and w = apply tablica (j-1, i)
                in
                  if u > w then
                    odtworz akumulator tx ciag_y (i-1) j
                  else
                    odtworz akumulator ciag_x ty i (j-1)
    in
      odtworz [] (List.reverse ciag_x) (List.reverse ciag_y)
              (List.length ciag_x) (List.length ciag_y);;
```

– Przykład.

15.3 Spamiętywanie

Mając do dyspozycji mapy możemy jeszcze prościej zapisać rozwiązanie, korzystając z techniki spamiętywania. Technika ta polega na pamiętaniu wszystkich obliczonych uprzednio rozwiązań podproblemów. Jest to skrzyżowanie rozwiązania rekurencyjnego i programowania dynamicznego.

W poniższym programie mapa przyporządkowuje parom liczb (i, j) pary (najdłuższy wspólny podciąg, jego długość). Wynikiem funkcji jest para (mapa, (najdłuższy wspólny podciąg dla (i, j) jak w parametrach, jego długość)).

```
let nwp ciag_x ciag_y =
  let rec buduj tab xl yl i j =
    let
      put tab a b = (update tab a b, b)
    in
      if dom tab (j, i) then
        (tab, apply tab (j, i))
      else
        (* trzeba polilczyć *)
        match xl with
          []      -> put tab (j, i) ([], 0) |
          (x::xt) ->
            match yl with
              []      -> put tab (j, i) ([], 0) |
              (y::yt) ->
                if x = y then
                  let (tab1, (w, d)) =  buduj tab xt yt (i-1) (j-1)
                  in put tab1 (j, i) (x::w, d+1)
                else
```

```
                let (tab1, (w1, d1)) = buduj tab xt yl (i-1) j
                in
                  let (tab2, (w2, d2)) = buduj tab1 xl yt i (j-1)
                  in
                    if d1 < d2 then
                      put tab2 (j, i) (w2, d2)
                    else
                      put tab2 (j, i) (w1, d1)
in
  let (_, (wynik, _)) = buduj empty_map ciag_x ciag_y
                              (List.length ciag_x) (List.length ciag_y)
  in
    wynik;;
```

15.4 Implementacja map

- Prosta:

```
exception Undefined;;

let apply m x = m x;;

let empty_map = function _ -> raise Undefined;;

let dom m x =
  try (m x); true
  with _ -> false ;;

let update f a v =
  function x -> if a = x then v else f x;;
```

- Drzewa BST — opowiedzieć idee operacji, bez implementacji. Koszt operacji — proporcjonalny do wysokości.
- Są lepsze struktury (drzewa zrównoważone) o czasie operacji $O(\log n)$.

15.5 Implementacja efektywna

- Struktura tablicy

```
type array = Element of (int * array * array * array);;

let rec zero = Element (0, zero, zero, zero);;
```

- Wypełnianie tablicy

```
let zbuduj_tablice ciag_x ciag_y =
  (* Dodaje kolejny wiersz tablicy,
```

```
    y= rozpatrywany element ciągu y.
  *)
  let dodaj_wiersz tablica y  =
    let rec dodawaj tablica lista =
      match lista with
        []     -> zero |
        (x::t) ->
          let Element (w, (Element (v, _, _, _) as tab), _, _) = tablica
          in
            let Element (u, _, _, _) as tab1 = dodawaj tab t
            in
              Element (dlugosc u v w x y, tab1, tab, tablica)
    in
      dodawaj tablica (List.reverse ciag_x)
  in
    (* Rekurencyjne budowanie tablicy. yl= pozostale elementy ciągu y. *)
    let rec buduj akumulator yl =
      match yl with
        []     -> akumulator |
        (y::t) -> buduj (dodaj_wiersz akumulator y) t
    in
      buduj 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
              let Element (_, _, tabv, _) = tab
              in
                odtworz (x::akumulator) tabv tx ty
            else
              let Element (_, tabu, _, tabw) = tab
              in
                let Element (u, _, _, _) = tabu
                and Element (w, _, _, _) = tabw
                in
                  if u > w then
                    odtworz akumulator tabu tx ciag_y
                  else
                    odtworz akumulator tabw ciag_x ty
```

```
in
  odtworz [] tablica (List.reverse ciag_x) (List.reverse ciag_y);;
```

Laboratorium

Zaimplementuj symulację problemu Juzefa Flawiusza. W symulacji wykorzystaj moduł implementujący listę cykliczną. Ustal zestaw operacji elementarnych na listach cyklicznych. Zaimplementuj listy cykliczne na dwa sposoby:

- najprościej jak potrafisz, choć niekoniecznie efektywnie,
- tak, aby operacje miały stały koszt (zamortyzowany).

Sprawdź, czy symulacja poprawnie współpracuje z obydwoma modułami. Jeśli do zaimplementowania list cyklicznych wykorzystujesz jakąś inną strukturę danych (np. z wykładu), to zamknij ją również wewnątrz modułu.

Ćwiczenia

- [III OI] Na szachownicy $n \times n$ należy ustawić n wież tak, żeby żadne dwie się nie szachowały, a ponadto każda z tych wież ma wyznaczony prostokąt, w którym ma stać. Znajdź takie ustawienie, lub stwierdź, że się nie da. (Wskazówka: problem sprowadza się do dwóch problemów jednowymiarowych.)
- Na ile sposobów można wydać n zł reszty przyjmując, że w obrocie są dostępne monety o zadanych (w postaci listy) nominałach.
- [CLR, ćw. 16-4] Firma planuje zorganizować przyjęcie. Hierarchia stanowisk w firmie ma strukturę drzewa (ogólnego). Każdy pracownik charakterycuje się pewną „towarzyskością" 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?

16 Backtracking

Nie zawsze da się zastosować programowanie zachłanne lub dynamiczne. Czasami wybór (optymalnego) rozwiązania w naturalny sposób sprowadza się do rozwiązań podproblemów, lecz struktura i liczba takich podproblemów ma charakter wykładniczy. Możemy wówczas starać się budować krok po kroku wszystkie możliwe rozwiązania, wycofując się z kroków, które nie prowadzą do rozwiązań. Istotne jet to, że w trakcie budowania rozwiązania okaże się, że częściowego rozwiązania nie możemy rozszerzyć, to wycofujemy się.

16.1 Problem ośmiu chetmanów

Jak ustawić n hetmanów na szachownicy $n \times n$ tak, aby żadne dwa się nie szachowały. 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 szachuje już ustawionych. Interesuje nas jeden wynik, który przekazujemy korzystając z mechanizmu wyjątków:

```
exception Rozwiazanie of ((int * int) list);;
```

Poniższe procedury sprawdzają możliwość dostawiania kolejnych hetmanów:

```
  (* Funkcja określająca, czy się szachują *)
  let szach (x1, y1) (x2, y2) =
    x1 = x2 or y1 = y2 or x1 + y1 = x2 + y2 or x1 - y1 = x2 - y2;;

  (* Czy można dostawić hetmana h do ustawionych już hetmanów het *)
  let rec mozna_dostawic h het =
    match het with
      []     -> true |
      (x::t) -> not (szach h x) && (mozna_dostawic h t);;
```

Sama rekurencyjna procedura wyszukująca ma nastąpującą postać:

```
let hetman n =
  (* Dostawia kolejnego hetmana, w k kolumnie i wierszach 1-l *)
  let rec dostaw het k l =
    if k = 0 then
      raise (Rozwiazanie het)
    else
      if l = 0 then
        (* Brak rozwiązania *)
        ()
      else
        let
          _ = if mozna_dostawic (k, l) het then
                dostaw ((k,l)::het) (k-1) n
              else
                ()
        in
```

```
        dostaw het k (l-1)
  in
    try
      let
        _ = dostaw [] n n
      in
        failwith "Brak rozwiązania"
    with
      Rozwiazanie x -> x;;
```

16.2 Obcinanie gałęzi

Zasadnicze znaczenie dla złożoności backtrackingu ma obcinanie gałęzi nie prowadzących do interesujących nas wyników. Zmniejszenie liczby rozgałęzień rekurencyjnych odpowiada zmniejszeniu podstawy potęgi będącej kosztem czasowym.

Możemy starać się obcinać takie 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 znalezionego już rozwiązania, to nie warto przeglądać takich gałęzi.

Warto wówczas zwrócić uwagę na kolejność rozpatrywania kolejnych wariantów. Należy najpierw wybierać takie, które mają szansę dać lepsze oszacowania.

Laboratorium

Zaimplementuj mapy jako drzewa BST.

Ćwiczenia

- Jaka jest minimalna liczba monet potrzebna do wydania n zł reszty?
- Znaleźć rozkład liczby na sumę minimalnej liczby liczb pierwszych.
- Problem obejścia szachownicy skoczkiem.
- Ustawić minimalną liczbę hetmanów szachujących wszystkie pola na szachownicy,

III OI Mokra robota. Dany jest zestaw naczyć szklanych o określonych objętościach, a także 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. (Nie koniecznie back-tracking, ale przeszukiwanie przestrzeni stanów.)

17 ALgorytm mini-max

Problem: jak zaprogramować optymalną strategię grania w grę dla dwóch graczy? Możemy tu zastosować technikę podobną do backtrackingu, 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 do wyboru, 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 ponaddto może być wieksza 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 przypisaliśmy 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 dobierać 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.

17.1 Algorytm mini-max

Sam algorytm postaramy się zapisać w sposób niezależny od rodzaju gry.

```
(* Gracz = +1/-1 odpowiednio max i min *)

(* Przeciwnik gracza who *)
let other who = -who;;

(* Ocena sytuacji s, gdy pierwszy ruch wykonuje who. *)
```

```
(* Wynikiem jest para (waga, ruch realizujący wagę). *)
let rec minimax s who =
  if finished s who then
    (* Gra skończona *)
    (result s who, pass)
  else
    (* Gra trwa *)
    let
      (* Funkcja przyporządkowująca ruchowi parę *)
      (* (waga sytuacji po * who, ruch). *)
      make_move m =
        let
          (w, _) = minimax (move s who m) (other who)
        in
          (who * w, m)
    in
      let
        (* waga optymalnego ruchu * who i optymalny ruch *)
        (w, m) = choose (map make_move (moves s who))
      in
        (w * who, m);;
```

Korzystamy tu z pomocniczej procedury wybierającej optymalny ruch:

```
(* Funkcja wybierająca z ciągu par (w, s) parę o maksymalnym w. *)
let choose l =
  let rec choose_rec (w, m) l =
    match l with
      []           -> (w, m) |
      ((w1, m1)::t) ->
        if w1 > w then
          choose_rec (w1, m1) t
        else
          choose_rec (w, m) t
  in
    match l with
      []     -> failwith "Pusta lista ruchów!"|
      (h::t) -> choose_rec h t;;
```

17.2 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.

```
(* Stan gry = liczba pozostałych kamieni *)
(* Ruch = liczba zdejmowanych kamieni. *)
(* Pas - brak ruchu, np. gdy gra skończona. *)
let pass = 0;;
```

```
(* Funkcja generująca dla danej sytuacji s i gracza who listę ruchów. *)
let moves s who =
  match s with
    0 -> [] |
    1 -> [1] |
    2 -> [1; 2] |
    _ -> [1; 2; 3];;

(* Pusta lista ruchów, to koniec gry. *)
let finished s who = moves s who = [];;

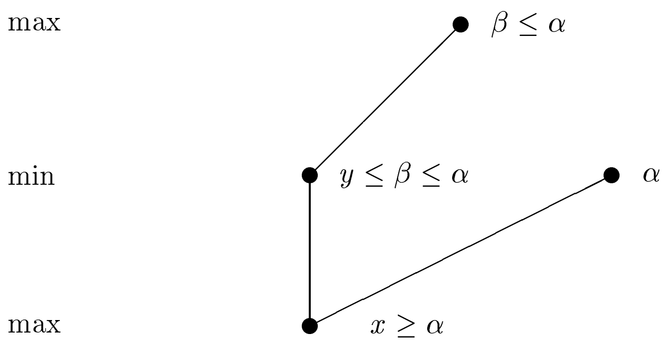
(* Funkcja przypisująca wagi liściom, s = sytuacja, who = gracz. *)
(* Zakładamy, że finished s who. *)
let result s who = other who;;

(* Rezultat wykonania ruchu m w sytuacji s przez gracza who *)
let move s who m = s - m;;
```

17.3 α-β obcięcie

Algorytm mini-max można polepszyć. Zauważmy, 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.

```
(* Algorytm alfa-beta obcięcia.                              *)
(* Ocena sytuacji s, gdy pierwszy ruch wykonuje who, a przeciwnik *)
(* jak dotąd uzyskał wagę alpha.                             *)
(* Wynikiem jest para (waga, ruch realizujący wagę).         *)
let rec alfabeta s who alpha =
  if finished s who then
    (* Gra skończona *)
    (result s who, pass)
  else
    (* Gra trwa *)

    (* Funkcja obliczająca wagę węzła,                             *)
    (* mv - lista ruchów do rozpatrzenia                           *)
    (* (beta, mb) - do tej pory obliczona waga i realizujący ją ruch *)
    (* (waga sytuacji po * who, ruch).                             *)
    let rec choose mv (beta, mb) =
      if beta * who >= alpha * who then
        (* Nie ma potrzeby dalszej analizy *)
        (beta, mb)
      else
        match mv with
          [] -> (beta, mb) |
          (m::t) ->
            let
              (w, _) = alfabeta (move s who m) (other who) beta
            in
              if w * who > who * beta then
                choose t (w, m)
              else
                choose t (beta, mb)
    in
      choose (moves s who) (no_result who, pass);;
```

Korzystamy tu z pewnych dodatkowych procedur pomocniczych:

```
(* Gracz główny. *)
let i = 1;;
```

```
(* Wynik gorszy od najgorszego - na potrzeby alfa-beta obcięcia. *)
let no_result who = -2 * who;;
```

17.4 Analia algorytmu α-β obcięcia

Efektywność tego algorytmu istonie 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 jest dokona obliczenia wszystkich podgałęzi. Natomiast w kolejnych wywołaniach rekurencyjnych mamy coraz większe szanse na obcinanie gałęci.

Załóżmy 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. Załóżmy, że na każdym poziomie mamy n gałęzi do rozpatrzenia. Oznaczmy gałęzie wychodzące z kolejnego wywołania rekurencyjnego (gracza grającego na min.) przez $B_1, \dots, B_n$. Załóżmy, że prawdopodobieństwo, iż $B_i > A$ jest równe p (niezależnie od i). Wówczas prawdopodobieństwo, że będziemy obliczać wagę i gałęzi wynosi

$$p^(k-1)$$

Tak więc oczekiwana liczba obliczanych wag gałęzi wynosi:

$$\sum_{k=1}^{n} kp^{k-1}(1-p) \leq (1-p)\sum_{k=0}^{\infty} kp^k = (1-p)\sum_{k=0}^{\infty}\sum_{i=k}^{\infty} p^i =$$

$$(1-p)\sum_{k=0}^{\infty}\left(p^k\sum_{i=0}^{\infty}p^i\right) = (1-p)\frac{1}{1-p}\sum_{k=0}^{\infty}p^k = \frac{1}{1-p}$$

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 $n = \frac{1}{2}$ daje nam $\sqrt{3n}$.

Oczywiście powyższe oszacowanie jest zgróbne. 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 zmienić istotnie asymptotyczną złożoność programu.

Laboratorium

Sortowanie topologiczne. Historyjka o budowie domu. Dane: liczba czynności i lista zależności. Wynik: przykładowa kolejność wykonania, lub wyjątek, gdy się nie da. Podpowiedzieć str. danych:

- dla każdej czynności mamy liczbę czynności, które mają być wykonane przed i listę czynności, które mają być wykonane po,
- czynności same są elementami kolekcji,
- dodatkowo mamy kolejkę FIFO z elementami, które mogą być wykonane pierwsze,
- początkowo przeglądamy kolekcję i wstawiamy takie elementy do kolejki,
- następnie usuwamy kolejne elementy z kolejki, zmniejszając liczniki czynnościom, jakie mają być wykonane po nich, a gdy licznik spada do zera, to wstawiamy je też do kolejki.

Zauważmy, że jedyne elementy dynamicznie się zmieniające to kolejka FIFO i liczniki czynności wykonywanych przed.

18 Programowanie imperatywne

18.1 Podstawowe instrukcje

- referencje, `let r = ref x;;`,
- wyłuskanie wartości, `!r;;`,
- dynamiczna zmiana wartości, `r := 5;;`
- kłopoty z polimorfizmem, referencja wskazuje na wartość przynajmniej tak ogólną jak typ referencji, wyłuskując wskazywaną wartość możemy ją ukonkretnić, przypisując wartość możemy podać wartość bardziej ogólną:

```
let f x y = x;;
let g x y = x+1;;
let h x y = x + y;;
let i x y = if y then x else 2 * x;;
let r = ref g;;
r := f;;
r := h;; ---zmienia się typ r!
r := i;; ---błąd
```

- średnik `;`,
- tablice `[| ...|]`, `Array.create długość wart.pocz.`, `a.(i) <- w`,
- pętla for: `for v = wyr to wyr do ...done`,
- pętla while: `while wyr do ...done`,
- wypisywanie: `print_...`,
- wczytywanie: `read_...`.

18.2 O tablicach

Np. heap sort na tablicy.

18.3 Funkcyjnie, czy imperatywnie

Kiedy programować funkcyjnie, a kiedy imperatywnie? Analiza programów funkcyjnych 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 programowaniu funkcyjnym.

Nie oznacza to jednak, że wszystko i zawsze należy programować funkcyjnie. Niektóre algorytmy (np. dynamiczne) dużo łatwiej jest zapisać z użyciem *imperatyenych struktur danych*, np. tablic. Należy je wówczas implementować imperatywnie. Nie przeszkadza to jednak takiemu ich zdefiniowaniu, aby cała imperatywność była ukryta przed użytkownikiem, stanowiła sekret implementacji.

Niektóre z kolei algorytmy łatwiej jest zapisać z użyciem *funkcyjnych struktur danych*, np. algebraicznych typów danych. Dotyczy to zwłaszcza wszelkiego rodzaju drzew i algorytmów rekurencyjnych. W takich przypadkach należy stosować programowanie funkcyjne.

W MLu odróżniamy funkcyjne i imperatywne struktury danych (do tych drugich należą referencje i tablice), co ułatwia rozdzielenie tych dwóch paradygmatów.

Osobną kategorię stanowią pliki, które ze swej natury są imperatywne. Jeśli nasz program ma operować na plikach, to jesteśmy skazani na (przynajmniej ukrytą) imperatywność.

Budowane przez nas systemy mają za cel modelowanie systemów istniejących w świecie rzeczywistym. Jedna z możliwych technik budowania takich systemów polega na stworzeniu obiektów obliczeniowych modelujących obiekty ze świata rzeczywistego. Tam obiekty charakteryzują się *stanem*, który może się *zmieniać* w czasie. Stan się zmienia, ale *tożsamość* obiektów pozostaje niezmienna. Tego aspektu nie jesteśmy w stanie modelować w programowaniu funkcyjnym. Możemy sledzić kolejne stany obiektu, ale za każdym razem będzie to dla nas inna wartość. I w takich właśnie przypadkach należy stosować programowanie imperatywne — aby modelować obiekty o niezmiennej tożsamości, ale zmieniającym się stanie.

18.4 Przykład: konto bankowe

Stan obiektów jest wyznaczany przez zawarte w nich *zmienne stanowe*. Zmienne takie możemy realizować w formie referencji.

```
let saldo = ref 0;;

let wplata kwota =
  if kwota > 0 then
    (saldo := !saldo + kwota;
    !saldo)
  else
    failwith "Z3a kwota";;

let wyplata kwota =
  if kwota > 0 then
    if kwota <= !saldo then
      (saldo := !saldo - kwota;
      !saldo)
    else
      failwith "Brak 6rodksw na koncie"
  else
    failwith "Z3a kwota";;
```

Jest to poprawna implementacja, choć nie najlepsza metodologicznie. Powinniśmy oddzielić *interfejs obiektu* złożony z metod zmiany jego stanu, od *implementacji* jego stanu za pomocą zmiennych stanowych. Najlepiej dodkonać tego wprowadzając odpowiednią barierę abstrakcji. Barierę taką możemy wprowadzić zamykając zmienne stanowe w pudełku, do którego dostęp mają tylko wybrane metody. Technikę taką nazywamy *hermetyzacją*.

Ponadto chcielibyśmy móc operować wieloma takimi obiektami, nie definiując wielu metod operowania.

```
let zaloz poczatek =
  let
    saldo = ref poczatek
  in
    let wplata kwota =
      if kwota > 0 then
        (saldo := !saldo + kwota;
        !saldo)
      else
        failwith "Zła kwota"
    and wyplata kwota =
      if kwota > 0 then
        if kwota <= !saldo then
          (saldo := !saldo - kwota;
          !saldo)
        else
          failwith "Brak środków na koncie"
      else
        failwith "Zła kwota"
    in
      (wplata, wyplata);;
```

Co by się stało gdyby nie było argumentu `poczatek`?

18.5 Przykład: metoda Monte Carlo

Metoda Monte Carlo jest to metoda przybliżania pewnych 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.

Oryginalnie metoda Monte Carlo powstała jako metoda przybliżania całek oznaczonych funkcji. 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, sprawdzna 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 kolo (x, y) = x *. x +. y *. 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 losowy_int =
  let stan = ref 0
  and a = 937126433
  and b = 937187
  in function () ->
    (
      stan := !stan * b + a;
      !stan
    );;

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 losowa_od_do od doo = losowa () *. (doo -. od) +. od;;
```

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.

18.6 Kilka słów o plikach

Pliki sekwencyjne, tekstowe i o dostępie swobodnym. Typy `in_channel` i `out_channel`. Operacje:

- `open_out[_bin]: string -> out_channel`,

- output_char: out_channel -> char -> unit,
- output_string: out_channel -> string -> unit,
- seek_out: out_channel -> int -> unit,
- pos_out: out_channel -> int,
- out_channel_length: out_channel -> int,
- close_out: out_channel -> unit,
- open_in[_bin]: string -> in_channel,
- input_char: in_channel -> char,
- input_line: in_channel -> string,
- seek_in: in_channel -> int -> unit i odpowiedniki,
- close_in: in_channel -> unit.

i wiele innych.

19 Moduły i hermetyzacja

Wprowadzane przez nas do tej pory bariery abstrakcji nie były wsparte żadnymi mechanizmami ze strony języka. Mechanizmy takie jednak istnieją. Są to moduły. Moduł to niezależny fragment programu, realizujący określone operacje i typy stanowiąze *interfejs* modułu. Sposób implementacji interfejsu stanowi sekret modułu i nie jest dostępny z zewnątrz.

Sam interfejs modułu to *sygnatura*. Zawiera ona:

- deklaracje typów:
  - abstrakcyjnych — bez żadnej definicji,
  - skrótowców — z podanym wyrażeniem typowym; realizacja typu musi być z nim zgodna, ale nie musi być tożsama,
  - typ algebraiczny — konstruktory są widoczne na zewnątrz; realizacja musi być taka sama,
- deklaracje wartości — w tym funkcji,
- deklaracje wyjątków.

Wszystko to są tylko deklaracje, a nie definicje. Opisują one co ma być stworzone, ale nie jak.

Sygnaturę zapisujemy następująco:

```
sig ...end
```

Jeśli chcemy zdefiniować i nazwać sygnaturę, to robimy to następująco:

```
module type Nazwa = ...;;
```

Deklarując moduł możemy podać sygnaturę, wówczas tylko te elementy, które są w sygnaturze będą widoczne. W przeciwnym przypadku, wszystko będzie widoczne.

Przykład:

```
module type S =
  sig
    type t;;
    type w = int;;
    val alst : t;;
    val mltp : t -> w;;
  end;;

module M: S =
  struct
    type t = int list;;
    type w = int;;
    let alst = [3; 5];;
    let rec mltp (l: t) =
      match l with
        [] ->  (1:w) |
        (h::t) -> h * (mltp t)
```

```
  end;;

M.alst;;
M.mltp M.alst;;

open M;;
mltp alst;;
```

Są również moduły sparametryzowane — funktory. Sygnatura funktora ma postać:

```
functor (nazwa modułu: sygnatura) ->sygnatura
```

Wówczas moduł takiego typu ma argumenty będące modułami.

Ćwiczenia

- rotacja cykliczna tablicy (w stałej pamięci dodatkowej, za pomocą odwracania fragmentów tablicy),
- prosta tablica hashująca (tablica list elementów),
- program usuwający komentarze z programu w OCAMLu.

Laboratorium

Należy napisać program, który wczyta z pliku o zadanej nazwie ciag słów (przez słowo rozumiemy tu maksymalny ciąg znaków różnych od spacji), a następnie wypisze do pliku o podanej nazwie wszystkie klasy abstrakcji relacji anagram określonej na tych slowach. Dwa slowa są w relacji anagram wtw. gdy jedno z nich można otrzymać z drugiego przestawiając w nim znaki (precyzyjniej: gdy multizbiory ich znaków są identyczne). Na przykład słowa abba i baba są anagramami, zaś ala i lala nie. Każda klasa abstrakcji powinna być wypisana w osobnej linii.

Przykladowy wynik pracy programu Dla danych:

```
ala  baba

   abba ala
lala
```

wynik powinien wyglądać następująco:

```
abba baba
ala
lala
```

Zalozenia dotyczace implementacji Program powinien składać się z trzech części:

- Modułu `Plik` — implementującego operacje na (jednym) pliku. Powinien on dostarczać następujące operacje:

- `otworz_plik: string -> bool` otwiera plik o podanej nazwie do czytania. Wynik informuje o tym, czy udało się otworzyć plik.
- `daj_slowo: unit -> string` daje kolejne słowo z otwartego pliku. Jeśli nie ma już słów daje słowo puste.
- `zamknij_plik: unit -> unit` zamyka otwarty plik.

• Modułu `Klasy` — implementującego operacje na strukturze danych przechowujacej klasy abstrakcji. Moduł ten powinien dostarczać następujące operacje:

  - `wstaw_slowo: string -> unit` — wstawia słowo do odpowiedniej klasy abstrakcji (ew. ja inicjalizując),
  - `wypisz_klasy: string -> unit` — wypisuje klasy abstrakcji do pliku o podanej nazwie,
  - `zapomnij: unit -> unit` — usuwa wszystkie klasy abstrakcji ze struktury.

  Jeśli zechcesz wydzielić moduły pomocnicze implementujące fragmenty struktury danych, to możesz tak zrobić.

• procedury `main` pytającej się użytkownika o nazwę pliku wejściowego i wyjściowego i wykonującej obliczenia.

Możesz założyć, że każde słowo mieści się w `stringu`. Nie zakładaj jednak nic o długości linii w pliku.

20 Strumienie

20.1 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, a nie koniecznie 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.

20.2 Czym jest strumień?

Formalnie strumień to ciąg — być może nieskończony. Pomysł polega jednak na tym, żeby był on „leniwy”, tzn. obliczane były tylko potrzebne wartości.

20.3 Uleniwianie

Model obliczeniowy MLa charakteryzuje się zachłannym obliczaniem wartości — argumenty funkcji są obliczane bez względu na to, czy są potrzebne. Jak można je uleniwić?

Oto proste podejście, polegające na zastąpieniu wartości funkcją zwracającą daną wartość.

```
type 'a delayed = unit -> 'a;;
let delay x = function () -> x;;
let force x = x ();;
let a = delay 4;;
force a;;
```

Jeśli faktycznie korzystamy z wartości argumentu, poganiamy go, żeby się obliczył. Jeśli jednak kilka razy potrzebujemy jego wartość, to obliczamy ją kilka razy, co jest nieefektywne. Oto ulepszona wersja (z zaszytą imperatywnością).

```
type 'a value = NoValue | Value of 'a;;
let memoize f =
  let
    v = ref NoValue
  in
    function () ->
      match !v with
        Value y -> y |
        NoValue ->
          let
            y = f ()
          in
            v := Value y;
            y;;
```

```
let delay x = memoize (delay x);;

let a = delay 4;;
force a;;

let b = memoize (function () -> print_int 4; 4);;
force b;;
force b;;
```

Zwróćmy uwagę na to, że jeżeli nie chcemy aby dana wartość była obliczana, to musimy ją zapisać w formie trywialnej funkcji przekazanej do `memoize`. W Ocamlu, moduł `Lazy` udiostępnia odpowiedni lukier syntaktyczny postaci:

```
lazy (print_int 4; 4) = memoize (function () -> print_int 4; 4)
```

20.4 Implementacja strumieni

Możemy teraz zaimplementować strumienie. Strumień to para: wartość i odroczony strumień pozostałych wartości.

```
(* Typ strumieniowy. *)
type 'a stream = EmptyStream |
                 Cons of ('a delayed * ('a stream delayed));;

(* Konstruktory i selektory strumieniowe *)
let empty (s: 'a stream) = s = EmptyStream;;


let head s =
  match s with
    Cons (h, _) -> force h;;

let tail s =
  match s with
    Cons (_, t) -> force t;;
```

Mając do dyspozycji takie konstruktory i selektory możemy zdefiniować kilka pomocniczych operacji na strumieniach:

```
(* Proste operacje na strumieniach *)
let rec const_stream c = Cons (lazy c, lazy (const_stream c));;

let ones = const_stream 1;;

let rec filter_stream p s =
  if empty s then
    EmptyStream
  else
```

```
    let h = head s
    and t = tail s
    in
      if p h then
        Cons
          (lazy h,
          lazy (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
    EmptyStream
  else
    Cons
      (lazy (f (head s)),
      lazy (stream_map f (tail s)));;

let rec for_each p s =
  if not (empty s) then begin
    p (head s);
    for_each p (tail s)
  end;;

let display_stream s =
  let p x =
    begin
      print_int x;
      print_string "\n";
      flush stdout
    end
  in
    for_each p s;;
```

Spróbujmy skonstruować nieskończony strumień liczb pierwszych, generowanych metodą sita Eratostenesa. (Schemat przepływu sygnałów.)

```
let rec integers_from x = Cons(lazy x, lazy (integers_from (x+1)));;

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 (lazy (head s), lazy (sieve  (sito (head s) (tail s))));;

let primes = sieve (integers_from 2);;
```

20.5 Schematy przepływu sygnałów

Parę słów o.

20.6 Definicje uwikłane

Analogia z równaniami różniczkowymi. Elementów danego strumienia można użyć do zdefiniowania jego samego.

```
let rec ones =  Cons (lazy 1, lazy ones);;

let rec add_streams s1 s2 =
  if empty s1 then
    EmptyStream
  else if empty s2 then
    EmptyStream
  else
    Cons (lazy (head s1 + head s2), lazy (add_streams (tail s1) (tail s2)));;

let rec integers =  Cons (lazy 1, lazy (add_streams ones integers));;
```

Przykład: strumień kolejnych liczb Fibonacciego.

```
(* Liczby Fibonacciego - nie uwik3ane *)
let fibs =
  let rec fibo a b =
    Cons
      (lazy (a + b),
      lazy (fibo b (a+b)))
  in fibo 0 1;;

(* Liczby Fibonacciego - uwik3ane *)
let rec fibs =
  Cons
    (lazy 0,
    lazy (Cons
      (lazy 1,
      lazy (add_streams  fibs (tail fibs)))
    ));;
```

Uwikłana definicja liczb pierwszych:

```
let square x = x * x;;

let rec primes =
   Cons (
     lazy 2,
     lazy (filter_stream prime (integers_from 3))
   )
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;;
\begin{verbatim}

Iteracje jako strumienie:
\begin{verbatim}
let sqrt_improve g x = (g +. x /. g) /. 2.0;;

let sqrt_stream x =
  let rec guesses =
    Cons (
      lazy 1.0,
      lazy (stream_map (function guess -> sqrt_improve guess x) guesses)
    )
  in guesses;;

let display_float_stream s =
  let p x =
    begin
      print_float x;
      print_string "\n";
      flush stdout
    end
  in
    for_each p s;;
```

Przybliżenia π.

```
let rec pi_summands n =
  let
    succ x =
      if x > 0.0 then
        -.x -. 2.0
      else
        -.x +. 2.0
```

```
  in
    Cons(
      lazy (1.0 /. n),
      lazy (pi_summands  (succ n))
    );;

let partial_sums s =
  let rec ps x s =
    if empty s then
      EmptyStream
    else
      Cons (
        lazy ((head s) +. x),
        lazy (ps ((head s) +. x) (tail s))
      )
  in
    ps 0.0 s
  ;;


let scale_stream c s =
  stream_map (function x -> x *. c) s;;

let pi_stream = scale_stream 4.0 (partial_sums (pi_summands 1.0));;
```

Przyspieszacze:

```
let square x = x *. x;;

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 (
      lazy (s2 -. (square (s2 -. s1)) /. (s0 -. 2.0 *. s1 +. s2)),
      lazy (euler_transform (tail s))
    );;

display_float_stream (euler_transform pi_stream);;

let rec make_tab t s =
  Cons (
    lazy s,
    lazy (make_tab t (t s))
  );;

let accelerate t s = stream_map head (make_tab t s);;
```

```
let pi_stream = accelerate euler_transform pi_stream;;
```

Ćwiczenia

Zdefiniuj strumienie i narysuj schematy odpowiadające tym definicjom:

- przeplot elementów dwóch strumieni (można sprytnie, zamieniając w wywołaniu rekurencyjnym miejscami argumenty),
- scalenie dwóch posortowanych strumieni
- strumień wszystkich par elementów z dwóch danych strumieni (w dowolnej kolejności).

Ciekawe zadania:

- pręt i łuk,
- wahania akcji na giełdzie,
- szeregowanie czynnośći o rosnących liniowo kosztach,
- stacje benzynowe z różnymi cenami.

Wyjątki

Przerywnik: trochę technikalii. Mechanizm wyjątków.

- Wyjątek – zaznaczenie sytuacji, gdy nie można zwrócić sensownej wartości. Sam wyjątek jest to wartość typu `exn` niosąca informację o przyczynie niepowodzenia.
- Podniesienie wyjątku powoduje przerwanie obliczeń – propaguje się na zewnątrz.
- Przechwycenie wyjątku – jeśli w danym wyrażeniu zostanie podniesiony wyjątek, to możemy go przechwycić i podać, jaka powinna być wartość całości w takim przypadku.
- ```
  Failure of string : exn        ---zdefiniowany standardowo
  exception Wyjatek of int;;      ---można deklarować własne

  Failure "kicha";;
  Wyjatek 4;;

  raise : exn ->α                ---podniesienie wyjątku
  try w:α with e:exn ->v:α
  { | e:exn ->v:α }*
  try raise (Wyjatek 2) with Wyjatek x ->2 * x;;
  ```

Dowodzenie poprawności

Elementy dowodzenia poprawności, z wieloma uproszczeniami.

Specyfikacja Specyfikacja – formuła opisująca jaką wartość chcemy uzyskać. Będziemy używać FOL, HOL i czasem ścisłego języka naturalnego.

Nieokreśloność Wartości mogą być nieokreślone – zapętlenie rekurencji, podniesienie wyjątku (pomijamy dokładnijszą obsługę wyjątków). Pomijamy przyczynę nieokreśloności. Nieokreśloność się propaguje. Wyrażenia logiczne też mogą być nieokreślone. Trzy rodzaje nieokreśloności:

- $==$ porównywane wartości są określone i są sobie równe,
- $\equiv$ porównywane wartości są określone i są sobie równe, lub są obydwie nieokreślone,
  ```

- $w\downarrow$ oznacza, że w jest określone,
- $=$ równość z programu ($x = y \Rightarrow x == y$).

Implementacja = Program.

Dowód poprawności Dowód, że wartości definiowane w implementacji spełniają specyfikację.

- Pamiętając o sposobie wiązania symboli z ich znaczeniami, możemy zastępować wartość jej definicją. (Nie będziemy formalizować tego wiązania. W rozpatrywanych przykładach będzie to zwykłe podstawienie. α-konwersja.)
- `if-then-else` trzy przypadki w zależności od warunku,
- Operacje logiczne – algorytmiczne znaczenie operatorów logicznych w logice trójwartościowej.
- λ-abstrakcja – zawsze jest określona.
- Aplikacja funkcji – określona, gdy argument i wyrażenie definiujące są określone.
- Chcąc udowodnić coś o wartościach zdefiniowanych rekurencyjnie musimy skorzystać z indukcji (po argumentach).
  - liczby naturalne – zwykła indukcja,
  - listy – indukcja po dlugości,
  - datatypy – indukcja strukturalna po wysokości drzewa,
  - ogólny przypadek – funkcja miary i indukcja po danych mających kolejne wartości funkcji miary.

  Definicje rekurencyjne są określone dla najmniejszego zbioru danych, dla którego mogą być określone (zgodnie z ich definicją). Powiedzieć trochę o monotoniczności konstrukcji programistycznych i istnieniu takiego zbioru, ale bez zbytniego gmatwania.
- Poprawność częściowa i całkowita – kwestia określoności. Jeżeli pominiemy funkcję miary, to mamy częściową poprawność.

Ekstensjonalność jeśli dwie funkcje są tak samo określone/nieokreślone, a tam, gdzie są określone mają takie wartości, to są sobie równe

$$\forall_{f,g} f\downarrow \wedge g\downarrow \wedge \forall_x fx \equiv gx \Rightarrow f == g$$

Dziel i zwyciężaj uogólnienie, wyłonienie mniejszego problemu(ów), dopasowanie do funkcji miary.

Przykłady Dla każdego przykładu: specyfikacja, implementacja i dowód.

- silnia zwykła,
- silnia z akumulatorem,
- wyszukiwanie elementu na liście – poprawność częsciowa,
- wyszukiwanie elementu na liście – poprawność pełna,
- mnożenie chłopów rosyjskich – poprawność częściowa,
- mnożenie chłopów rosyjskich – poprawność pełna.
- merge sort.