Wstęp do programowania i Metody programowania (potok funkcyjny)

Marcin Kubica

2006/2007

Spis treści

Kilka słów na początek

0.1 Programowanie funkcyjne vs. imperatywne

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

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

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

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

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

0.2 Dwa potoki: imperatywny i funkcyjny

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

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

0.3 Kwestie organizacyjne

- Zasady zaliczenia — 3 kolokwia + ew. egzamin pisemny.
- Po wyborze potoku nie będzie możliwości przeniesienia się.
- Test wstępny — kwalifikacja na potok funkcyjny i „poziomowanie" grup imperatywnych.
- Zadania z kolokwiów i egzaminów z lat poprzednich zostały umieszczone w niniejszym skrypcie, choć nie zostało to w żaden szczególny sposób zaznaczone.

0.4 Podziękowania

Niniejsze materiały powstały na podstawie notatek do prowadzonych przeze mnie, na przestrzeni kilku lat, wykładów ze Wstępu do programowania i Metod programowania (potok funkcyjny). Chciałbym gorąco podziękować moim kolegom, którzy w tym czasie prowadzili ćwiczenia z tych przedmiotów: Jackowi Chrząszczowi, Bogusiowi Kluge, Mikołajowi Konarskiemu, Łukaszowi Lwowi, Robertowi Maronowi i Kubie Pawlewiczowi. Ich uwagi i propozycje miały wpływ na postać prowadzonych zajęć, a więc i również na te materiały. W szczególności część zamieszczonych tu zadań pochodzi od nich.

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

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

0.5 Literatura

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

Wykład 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 *Ocaml*. Nie jest to jednak kurs programowania w tym języku, lecz kurs podstawowych zasad formułowania zaklęć dla procesów obliczeniowych.

1.2 Kilka podstawowych pojęć

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

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

1.3 Algorytm

Muhammad Ibn Musa Al-Chuwarizmi ok. 780–850 r. n.e. matematyk, geograf i astronom działający w Bagdadzie na dworze Haruna Al-Raszida. Autor:

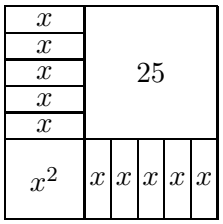
- *Tablic* (astronomia, kalendarz, funkcje trygonometryczne),
- *Arytmetyki* (hinduski zapis pozycyjny, metody rachunkowe),
- *Algebry* (równania liniowe i kwadratowe),
- *Kalendarza żydowskiego*, *Kronik*, *Geografii* i *Astrologii.*

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.

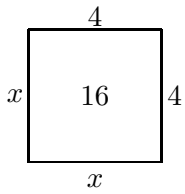
a	b	c	równanie	przykład	komentarz
$+$	$+$	$+$	$ax^2+bx+c=0$	—	—
$+$	$+$	0	$ax^2+bx=0$	—	—
$+$	$+$	$-$	$ax^2+bx=c$	$x^2+10x=39$	metoda 1
$+$	0	$+$	$ax^2+c=0$	—	—
$+$	0	0	$ax^2=0$	—	—
$+$	0	$-$	$ax^2=c$	$5x^2=80$	metoda 2
$+$	$-$	$+$	$ax^2+c=bx$	$x^2+21=10x$	metoda 3
$+$	$-$	0	$ax^2=bx$	$x^2=5x$	metoda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

Metoda 1: Metodę tę ilustruje następujący rysunek:

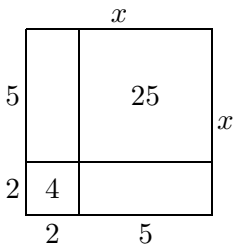


Pole dużego kwadratu wynosi $39+25=64=8\times 8$. Stąd $x+5=8$, czyli $x=3$.

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



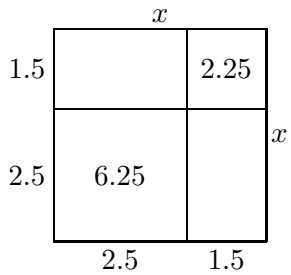
Metoda 3: Metoda ta stanowi, w pewnym sensie, odwrócenie metody 1. Poniższy rysunek ilustruje jej zastosowanie do przykładowego równania:



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

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

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



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

1.4 Inne języki programowania

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

- Przepisy kulinarne:
  - zamówienie — specyfikacja,
  - przepis — program,
  - gotowanie — wykonanie,

  - potrawa — efekt,
  - smakuje? — weryfikacja.

- 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.

  Oto definicje i użycie tabelki, która sama się wypełnia liczbami Fibonacciego:

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

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

\fibtab{12}
```

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

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

- Opisy skalowalnych czcionek (np. MetaFont):
  - opis czcionek — program,
  - generowanie wyglądu czcionek przy zadanej rozdzielczości i dla 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 $\langle R, +, -, *, /, \sqrt{}, <, = \rangle$. W ogólności możemy mieć kilka zbiorów elementarnych wartości, np. liczby całkowite, rzeczywiste, znaki, wartości logiczne.

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

Laboratorium

Zapoznanie się ze środowiskiem:

- logowanie się,
- uruchamianie przeglądarki i konsoli/terminala,
- `mc`,
- edytory: `mcedit`, `vi`, `emacs`, `gedit`, `kate`, `joe`,
- podstawowe polecenia: `ls`, `cd`, slashologia, `mkdir`, `pwd`, `mv`, `rm`, semantyka `?` i `*`, `chmod`,
- procesy: `ps`, `top`, uruchamianie w tle (`&`), `Ctrl-Z`, `Ctrl-C`, `jobs`, `bg`, `fg`, `kill`, `killall`,
- przekierowanie wejścia/wyjścia: `>`, `»`, `<`, `|`, `‘...‘`, `cat`, `echo`, `more`, `less`, `grep`, `wc`, `sort`, `uniq`,
- zmienne: `ZMIENNA=...`, `$ZMIENNA`, `${ZMIENNA}`,
- pętla `for`.

Ćwiczenia

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

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

Wytyczne dla prowadzących ćwiczenia

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

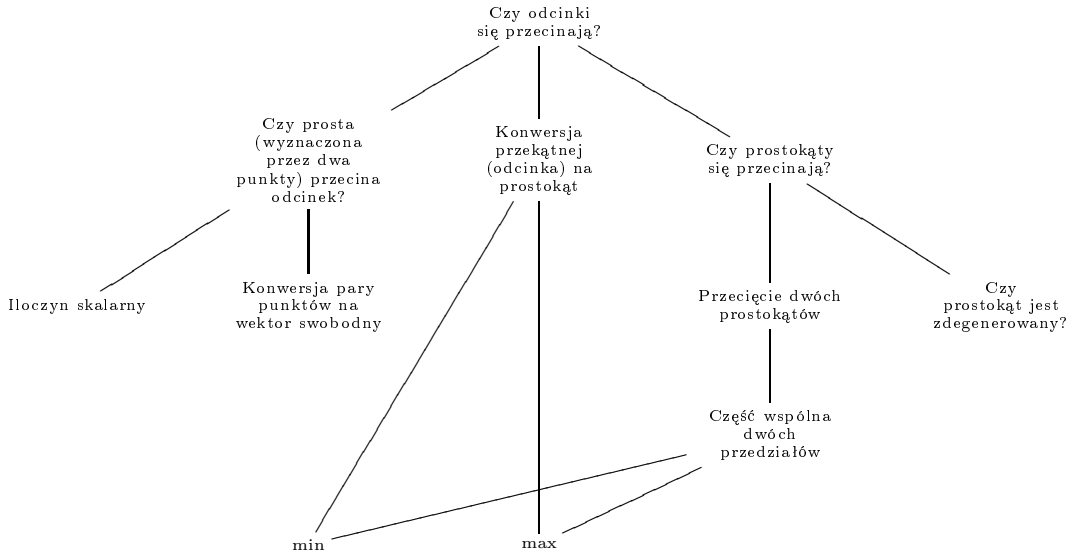
- Reprezentację danych. W treściach zadań nie jest określony sposób reprezentacji punktów, odcinków, prostokątów itp. Wybór reprezentacji jest prosty, ale należy go dokonać. Czasami ma istotny wpływ na długość rozwiązania.
- Podział problemu na podproblemy, czyli jakie funkcje pomocnicze będą powstawać. Dobrym podziałom odpowiadają prostsze rozwiązania. W trakcie rozwiązywania można dyskretnie nadzorować ten proces, a na koniec ćwiczeń należy na to zwrócić uwagę.

Uwagi dotyczące zadań:

Ad. 1 Punkt to para współrzędnych. Prostokąt to para jego wierzchołków (dolny lewy i górny prawy) lub para zakresów. Należy zwrócić uwagę na to, że przecięcie dwóch prostokątów może być puste. Najlepiej przyjąć, że prostokąty zdegenerowane są dopuszczalne i reprezentują pusty zbiór. Podział na podproblemy: rzutowanie, przecięcie odcinków na osi, budowa prostokąta na podstawie jego rzutów.

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

Możliwy podział na podproblemy:



Ad. 3 Równoległobok to punkt i dwa wektory swobodne. Równoległoboki się przecinają jeżeli:

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

Możliwy podział na podproblemy: suma wektorów i przesuwanie punktów, iloczyn wektorowy, czy punkt jest wewnątrz równoległoboku, czy odcinek przecina przekątną (ew. obwód) równoległoboku, czy przekątne (lub ew. obwody) dwóch równoległoboków się przecinają.

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

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

Wykład 2. Podstawy języka programowania

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

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

Nasza dziedzina algorytmiczna zawiera m.in.:

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

Powtórka: rozróżnienie symboli od ich interpretacji.

2.1 BNF

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

- W BNF-ie definiujemy możliwe postaci rozmaitych *konstrukcji składniowych*. Konstrukcje te mają nazwy postaci: ⟨`konstrukcja`⟩. Odpowiadają one nieterminalom w gramatykach bezkontekstowych.
- Opis składni składa się z szeregu reguł postaci:

  $$\langle \texttt{konstrukcja} \rangle ::= \quad \textit{możliwa postać}$$

  Reguły te opisują możliwe postaci poszczególnych konstrukcji składniowych. Jeżeli dla danej konstrukcji mamy wiele reguł, to traktujemy je jak alternatywy. Reguły te odpowiadają produkcjom w gramatykach bezkontekstowych.
- Po prawej stronie reguł możemy używać następujących konstrukcji:
  - podając tekst, który ma się pojawić dosłownie, będziemy go podkreślać, na przykład: `słowo kluczowe` ,
  - nazw konstrukcji składniowych postaci ⟨`konstrukcja`⟩ możemy używać również po prawej stronie reguł, na oznaczenie miejsc, w których się one pojawiają,
  - chcąc opisać alternatywne postaci danej konstrukcji używamy pionowej kreski, $\ldots|\ldots$,

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

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

Tego formalizmu będziemy używać do opisu składni.

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

```
⟨adres⟩        ::= ⟨adresat⟩ , ⟨adres lokalu⟩ , ⟨adres miasta⟩ [ , ⟨adres kraju⟩ ]
⟨adresat⟩      ::= [ W.P. | Sz.Pan. ] ⟨napis⟩
⟨adres lokalu⟩ ::= ⟨ulica⟩ ⟨numer⟩ [ / ⟨numer⟩ ] [ m ⟨numer⟩ | / ⟨numer⟩ ]
⟨adres miasta⟩ ::= [ ⟨kod⟩ ] ⟨napis⟩
⟨adres kraju⟩  ::= ⟨napis⟩
⟨kod⟩          ::= ⟨cyfra⟩ ⟨cyfra⟩ - ⟨cyfra⟩ ⟨cyfra⟩ ⟨cyfra⟩
⟨cyfra⟩        ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨numer⟩        ::= ⟨cyfra⟩ [ ⟨numer⟩ ]
```

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

2.2 Przyrostowy tryb pracy

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

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

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

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

```
⟨program⟩ ::= { ⟨jednostka kompilacji⟩ ;; }*
```

2.3 Wyrażenia

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

Przykład:

```
42;;
- : int = 42

36 + 6;;
- : int = 42

3 * 14;;
- : int = 42

100 - 58;;
- : int = 42

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

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

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

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

Wszystko, czego potrzeba do budowy wyrażeń, to:

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

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

2.3.1 Składnia wyrażeń

⟨jednostka kompilacji⟩ ::= ⟨wyrażenie⟩ | ...
⟨wyrażenie⟩ ::= (⟨wyrażenie⟩) | { ⟨wyrażenie⟩ }$^+$ |
⟨operator unarny⟩ ⟨wyrażenie⟩ |
⟨wyrażenie⟩ ⟨operator binarny⟩ ⟨wyrażenie⟩ |
⟨identyfikator⟩ | ⟨stała całkowita⟩ |
⟨stała zmiennopozycyjna⟩ | ⟨stała napisowa⟩ | ...
⟨operator unarny⟩ ::= - | not | ...
⟨operator binarny⟩ ::= + | +. | - | -. | * | *. | / | /. | mod | || | && |
= | < | <= | > | >= | ^ | ...

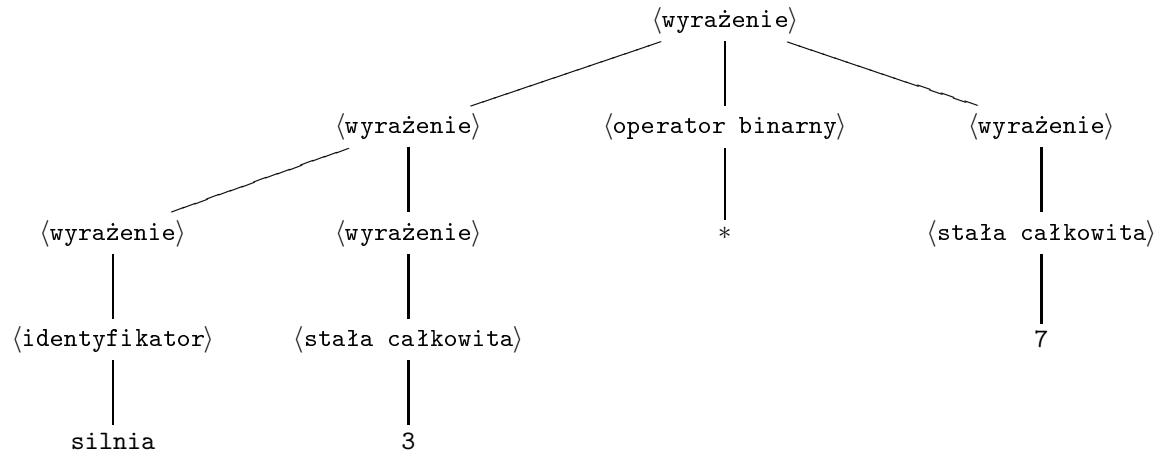
Zastosowania procedur wiążą najsilniej, operatory unarne słabiej, a operatory binarne najsłabiej, przy czym zachowana jest tradycyjna kolejność wiązania operacji arytmetycznych.

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

2.3.2 Obliczanie wartości wyrażeń

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

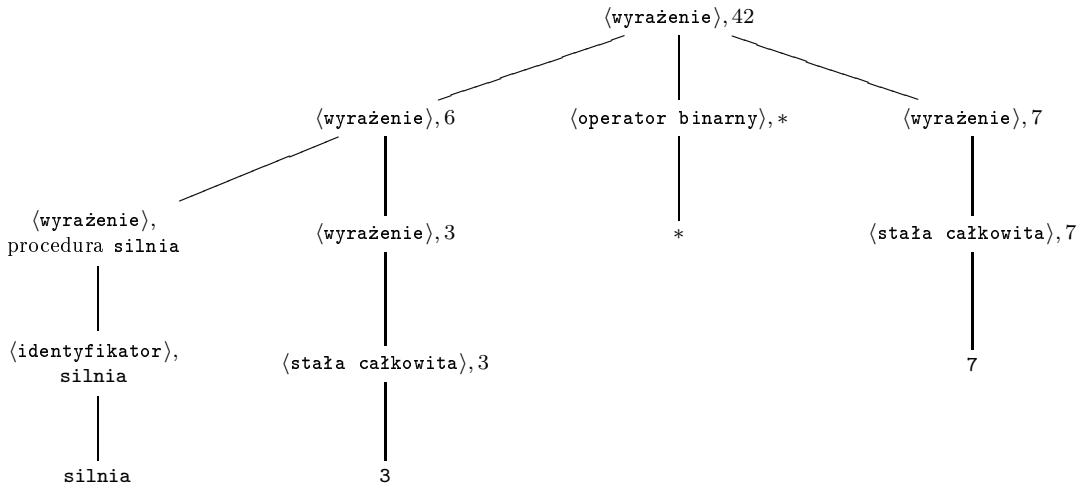
Przykład: Oto drzewo wyprowadzenia dla wyrażenia `silnia 3 * 7`:



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

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

Przykład: Oto powyższe drzewo wyprowadzenia udekorowane wartościami:



2.4 Definicje stałych

Kolejną podstawową jednostką kompilacji jest definicja stałej.

⟨jednostka kompilacji⟩ ::= ⟨definicja⟩ | ...
⟨definicja⟩ ::= let ⟨identyfikator⟩ = ⟨wyrażenie⟩ | ...

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

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

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

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

Przykład:

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

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

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

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

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

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

⟨definicja⟩ ::= let ⟨identyfikator⟩ = ⟨wyrażenie⟩
{ and ⟨identyfikator⟩ = ⟨wyrażenie⟩ }*

Przykład:

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

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

2.5 Wartości logiczne i wyrażenia warunkowe.

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

⟨wyrażenie⟩ ::= if ⟨wyrażenie⟩ then ⟨wyrażenie⟩ else ⟨wyrażenie⟩

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

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

Przykład:

```
let x  = 0.0238095238095238082;;
val x : float = 0.0238095238095238082

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

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

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

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

2.6 Definicje procedur

Mamy tylko trzy podstawowe konstrukcje językowe:

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

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

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

Definicje nazwanych procedur mają następującą postać:

⟨definicja⟩ ::= let ⟨identyfikator⟩ { ⟨identyfikator⟩ }$^+$ = ⟨wyrażenie⟩

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

Przykład:

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

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

```
val square : float -> float = <fun>

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

pow 4.0;;
- : float = 50.24

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

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

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

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

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

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

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

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

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

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

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

2.6.1 λ-abstrakcja

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

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

$$\langle \texttt{wyrażenie} \rangle ::= \quad \underline{\texttt{function}}\ \langle \texttt{identyfikator} \rangle\ \underline{\texttt{->}}\ \langle \texttt{wyrażenie} \rangle$$

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

Procedury wieloargumentowe zapisujemy w postaci:

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

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

$$\langle \texttt{wyrażenie} \rangle ::= \quad \underline{\texttt{fun}}\ \{\, \langle \texttt{identyfikator} \rangle \,\}^{+}\ \underline{\texttt{->}}\ \langle \texttt{wyrażenie} \rangle$$

Przykład: Oto alternatywna definicja procedur z poprzedniego przykładu:

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

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

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

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

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

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

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

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

2.6.2 Procedury rekurencyjne

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

⟨definicja⟩ ::= let rec { ⟨identyfikator⟩ }$^+$ = ⟨wyrażenie⟩

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

Przykład: Przykłady definicji rekurencyjnych:

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

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

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

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

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

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

```
let rec x = x + 2;;
This kind of expression is not allowed as right-hand side of 'let rec'
```

Istnieją inne obiekty rekurencyjne, niż procedury.

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

2.6.3 Rekurencja ogonowa

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

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

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

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

Przykład: Silnia z akumulatorem:

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

let silnia x = s 1 x;;
val silnia : int -> int = <fun>

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

Przykład: Liczby Fibonacciego z akumulatorem:

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

let fib n = fibpom 0 1 n;;
```

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

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

2.7 Definicje lokalne

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

⟨wyrażenie⟩ ::= ⟨definicja⟩ in ⟨wyrażenie⟩

Zakres definicji jest ograniczony do wyrażenia po `in`.

Przykład:

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

pow 4.0;;
- : float = 50.24

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

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

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

2.8 Komentarze

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

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

Ćwiczenia

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

1. Stopień parzystości liczby całkowitej x, to największa taka liczba naturalna i, że x dzieli się przez 2^i. Liczby nieparzyste mają stopień parzystości 0, liczby 2 i -6 mają stopień parzystości 1, a liczby 4 i 12 mają stopień parzystości 2. Przyjmujemy, że 0 ma stopień parzystości -1.

   Napisz procedurę `parzystość` wyznaczającą stopień parzystości danej liczby całkowitej.

2. Napisz procedurę, która przekształca daną liczbę naturalną w taką, w której cyfry występują w odwrotnej kolejności, np. 1234 jest przekształcane na 4321.

3. Sumy kolejnych liczb nieparzystych dają kwadraty kolejnych liczb naturalnych, zgodnie ze wzorem: $\sum_{i=1}^{k}(2i-1) = k^2$. Wykorzystaj ten fakt do napisania procedury `sqrt` obliczającej `sqrt x` $= \lfloor\sqrt{x}\rfloor$ i nie korzystającej z mnożenia, ani dzielenia.

4. Napisz procedurę, która sprawdza, czy dana liczba jest pierwsza.

5. Napisz procedurę, która sprawdza, czy dana liczba jest podzielna przez 9 w następujący sposób: jedyne liczby jednocyfrowe podzielne przez 9 to 9 i 0; reszta z dzielenia liczby wielocyfrowej przez 9 jest taka sama, jak reszta z dzielenia sumy jej cyfr przez 9; jeśli suma cyfr jest wielocyfrowa, to całość powtarzamy, aż do uzyskania liczby jednocyfrowej.

6. Napisz procedurę, która sprawdza czy dana liczba jest podzielna przez 11 w następujący sposób: sumujemy cyfry liczby znajdujące się na parzystych pozycjach, oraz te na nieparzystych pozycjach, różnica tych dwóch liczb przystaje modulo 11 do wyjściowej liczby; krok ten należy powtarzać aż do uzyskania liczby jednocyfrowej.

7. Zaimplementuj kodowanie par liczb naturalnych jako liczby naturalne. To znaczy, napisz procedurę dwuargumentową, która koduje dwie liczby dane jako argumenty w jednej liczbie naturalnej. Dodatkowo napisz dwie procedury, które wydobywają z zakodowanej pary odpowiednio pierwszą i drugą liczbę.

8. Napisz procedurę, która dla danej liczby n sprawdzi czy pierścień reszt modulo n zawiera nietrywialne pierwiastki z 1 (tj. takie liczby k, $k \neq 1$, $k \neq n-1$, że $k^2 \equiv 1 \bmod n$). Nota bene, jeśli takie pierwiastki istnieją, to liczba n nie jest pierwsza. Odwrotna implikacja jednak nie zachodzi — np. dla $n = 4$ nie ma nietrywialnych pierwiastków z 1.

Wytyczne dla prowadzących ćwiczenia

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

Wykład 3. Dekompozycja problemu, weryfikacja rozwiązania

3.1 Specyfikacja problemu

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

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

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

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

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

- Tworząc procedury pomocnicze powinniśmy określić również ich specyfikacje. W przypadku iteracyjnych procedur rekurencyjnych są to niezmienniki iteracji.
- Schemat dowodzenia własności stopu:
  - określamy funkcję miary, dowód przebiega indukcyjnie (po wartościach funkcji miary),
  - zakładamy, że argumenty wywołania spełniają warunek początkowy, dla danych spełniających warunek początkowy wartość funkcji miary jest nieujemna,
  - pokazujemy, że wywołania procedur spełniają odpowiednie warunki początkowe,
  - dla wszystkich wywołań rekurencyjnych pokazujemy, że funkcja miary maleje.
- Schemat dowodzenia częściowej poprawności:
  - zakładamy, że argumenty spełniają warunek początkowy, a wszystkie wywołania procedur (łącznie z rekurencyjnymi) spełniają specyfikację,
  - pokazujemy, że wynik jest zgodny z warunkiem końcowym.

3.2 Euklides (wersja z odejmowaniem)

Specyfikacja: dla $n, m > 0$ mamy (`nwd` m n) $= NWD(m, n)$. implementacja:

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

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

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

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

3.3 Metoda Newtona liczenia pierwiastków

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

Specyfikacja `sqrt` x:

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

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

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

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

- warunek wstępny: $x \geq 0$, $g \geq 0$,
- `good` $g\ x \Rightarrow |g - \sqrt{x}| \leq \varepsilon$

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

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

Stąd:

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

Czyli:

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

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

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

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

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

$$\exists_{n \in \mathbb{N}}\, \texttt{good}\left((\texttt{fun}\ g \rightarrow \texttt{improve}\ g\ x)^n g\right)$$

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

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

Warunek wstępny tak zaimplementowanej procedury `improve`, to:

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

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

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

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

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

Zweryfikujemy teraz własność stopu naszego algorytmu. Zauważmy, że kolejne przybliżenia pierwiastka są dodatnie. Zaczynamy od 1, a każdy kolejny krok przekształca dodatnie przy-

bliżenie w dodatnie. Błąd kolejnego przybliżenia wynosi:

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

Rozważmy przypadki:

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

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

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

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

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

Stąd:

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

Czyli:

$$|g^2-x|\le\varepsilon\cdot g \;\Rightarrow\; \texttt{good}\ g\ x$$

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

3.4 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`,
- podnoszenie do kwadratu: `square`,
- średnia: `average`,
- wartość bezwzględna: `abs`.

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

- Należy podzielić problem trudniejszy na kilka prostszych „zgodnie z jego strukturą", tzn. tak, aby można było prosto wyrazić rozwiązanie całego problemu za pomocą rozwiązań podproblemów — w ten sposób powstające definicje są proste i łatwe do ogarnięcia.
- Ponadto należy tak formułować podproblemy, aby były jak najbardziej ogólne — wówczas w większych systemach istnieje możliwość wykorzystania jednego rozwiązania w kilku miejscach.

Zasada pobożnych życzeń:

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

Kilka innych dobrych zasad:

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

Ćwiczenia

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

1. Udowodnij, że dla każdego naturalnego n zachodzi `fib` $n = Fib_n$. Podaj specyfikację dla `fibpom` i udowodnij ją przez indukcję.

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

2. Potęgowanie liczb całkowitych (z akumulatorem).

3. Napisz procedurę obliczającą $\lfloor\sqrt{n}\rfloor$, tym razem przez bisekcję, dzieląc przedział, w którym znajduje się wynik. Dowodząc poprawności zwróć uwagę na warunki brzegowe.

4. Dana jest funkcja $f : \{1, 2, \ldots, n\} \to \mathcal{Z}$. Należy znaleźć takie $1 \leq a \leq b \leq n$, że suma $\Sigma_{i=a}^{b} f(i)$ jest największa. Napisz taką procedurę `p`, że `p` f $n = b - a + 1$ (dla a i b jak wyżej).

5. Dana jest funkcja nierosnąca f : `int` $\to$ `int`, taka, że $f(x + 1) \geq f(x) - 1$. Napisz procedurę, która znajduje punkt stały tej funkcji, tzn. taką liczbę x, że $f(x) = x$ (lub jeśli punkt stały nie istnieje, to taką liczbę x, że $f(x - 1) > x$ oraz $f(x) \leq x$).

6. Dana jest różnowartościowa funkcja f : `int` $\to$ `int`. Napisz procedurę `maksima`, która dla danych liczb l i p ($l \leq p$) obliczy ile lokalnych maksimów ma funkcja f w przedziale $[l, p]$.

7. Przypomnij sobie rozwiązania zadań z poprzedniego wykładu. Jeśli rozwiązałeś je nie stosując rekurencji ogonowej, to czy potrafisz je rozwiązać (lepiej) stosując ją?

Wytyczne dla prowadzących ćwiczenia

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

Uwagi do zadań:

Ad. 2 To zadanie należy rozwiązać w dwóch wersjach: najpierw prościej (liniowa złożoność czasowa), a następnie bardziej efektywnie (logarytmiczna złożoność czasowa). W obu przypadkach ogonowo.

Ad. 4 To zadanie można rozwiązać w liniowej złożoności czasowej. Jeżeli studenci nie zauważą tego od razu, należy najpierw rozwiązać je mniej efektywnie (i w bardziej skomplikowany sposób).

Ad. 5 Efektywne rozwiązanie tego zadania jest bardziej skomplikowane niż proste nieefektywne rozwiązanie. Można zrobić oba warianty.

Ad. 7 „Lepiej", to bardziej efektywnie.

Wykład 4. Struktury danych

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

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

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

4.1 Typy wbudowane

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

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

4.2 Konstruktory

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

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

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

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

4.3 Wzorce

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

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

$$\langle \texttt{wzorzec} \rangle ::= \quad \langle \texttt{identyfikator} \rangle \mid \underline{\texttt{\_}} \mid \langle \texttt{stała} \rangle \mid \ldots$$

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

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

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

$$\begin{array}{lll}
\langle \texttt{definicja} \rangle & ::= & \underline{\texttt{let}}\ \langle \texttt{wzorzec} \rangle\ \underline{\texttt{=}}\ \langle \texttt{wyrażenie} \rangle\ \mid \\
& & \underline{\texttt{let}}\ [\ \underline{\texttt{rec}}\ ]\ \langle \texttt{identyfikator} \rangle\ \{\ \langle \texttt{wzorzec} \rangle\ \}^*\ \underline{\texttt{=}}\ \langle \texttt{wyrażenie} \rangle \\
\langle \texttt{wyrażenie} \rangle & ::= & \underline{\texttt{match}}\ \langle \texttt{wyrażenie} \rangle\ \underline{\texttt{with}}\ \{\ \langle \texttt{wzorzec} \rangle\ \underline{\texttt{->}}\ \langle \texttt{wyrażenie} \rangle\ \underline{\texttt{|}}\ \}^* \\
& & \langle \texttt{wzorzec} \rangle\ \underline{\texttt{->}}\ \langle \texttt{wyrażenie} \rangle\ \mid \\
& & \underline{\texttt{function}}\ \{\ \langle \texttt{wzorzec} \rangle\ \underline{\texttt{->}}\ \langle \texttt{wyrażenie} \rangle\ \underline{\texttt{|}}\ \}^* \\
& & \langle \texttt{wzorzec} \rangle\ \underline{\texttt{->}}\ \langle \texttt{wyrażenie} \rangle
\end{array}$$

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

Przykład:

```
3;;
- : int = 3

let _ = 3;;
- : int = 3
```

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

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

Przykład:

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

f 6 9;;
- : int = 42

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

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

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

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

Przykład:

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

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

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

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

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

4.4 Produkty kartezjańskie

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

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

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

⟨wyrażenie⟩ ::= ([⟨wyrażenie⟩ { , ⟨wyrażenie⟩ }*])
⟨wzorzec⟩ ::= ([⟨wzorzec⟩ { , ⟨wzorzec⟩ }*])

Przykład:

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

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

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

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

```
let para x y = (x, y);;
val para : 'a -> 'b -> 'a * 'b = <fun>

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

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

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

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

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

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

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

4.5 Listy

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

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

Chcąc podać listę n-elementową nie musimy pisać: $x_1 :: (x_2 :: \cdots :: (x_n :: [])\ldots)$, lecz możemy użyć skrótu notacyjnego: $[x_1; x_2; \ldots; x_n]$. Dostępna jest też operacja sklejania list `@`.

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

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

Przykład:

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

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

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

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

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

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

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

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

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

Przykład: Definicje list nieskończonych:

```
let rec jedynki = 1::jedynki;;
```

```
val jedynki : int list = [1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
1; ...]

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

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

cykl @ [1;2;3];;
Stack overflow during evaluation (looping recursion?).
```

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

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

Przykład:

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

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

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

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

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

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

4.6 Deklaracje typów

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

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

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

Przykład:

```
type p = int * float;;

type 'a lista = 'a list;;

type ('a, 'b) para = 'a * 'b;;

type 'a t = ('a, 'a) para * ('a list) list;;
```

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

⟨wyrażenie⟩ ::= (⟨wyrażenie⟩ : ⟨typ⟩)
⟨wzorzec⟩ ::= (⟨wzorzec⟩ : ⟨typ⟩)

Przykład: Specyfikacja typów.

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

type vector = point;;
type vector = point

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

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

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

4.7 Rekordy

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

⟨typ⟩ ::= { { ⟨identyfikator⟩ : ⟨typ⟩ ; }* ⟨identyfikator⟩ : ⟨typ⟩ [;] }

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

⟨wyrażenie⟩ ::= { { ⟨identyfikator⟩ = ⟨wyrażenie⟩ ; }*
⟨identyfikator⟩ = ⟨wyrażenie⟩ [;] }
⟨wzorzec⟩ ::= { { ⟨identyfikator⟩ = ⟨wzorzec⟩ ; }*
⟨identyfikator⟩ = ⟨wzorzec⟩ [;] }

Używając takiego konstruktora do budowy rekordu musimy podać wartości wszystkich pól (w dowolnej kolejności). Natomiast używając go jako wzorca, możemy podać tylko interesujące

nas pola. Do pojedynczych pól rekordów możemy się również odwoływać tak, jak w innych językach programowania, podając nazwę pola po kropce.

⟨wyrażenie⟩ ::= ⟨wyrażenie⟩ . ⟨identyfikator⟩

Przykład:

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

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

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

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

q.licznik;;
- : int = 3
```

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

4.8 Typy wariantowe

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

⟨typ⟩ ::= ⟨wariant⟩ { | ⟨wariant⟩ }*
⟨wariant⟩ ::= ⟨Identyfikator⟩ [of ⟨typ⟩]

Przez ⟨Identyfikator⟩ oznaczamy identyfikatory rozpoczynające się wielką literą. Natomiast ⟨identyfikator⟩ oznacza identyfikatory rozpoczynające się małą literą. Jak zobaczymy za chwilę, rozróżnienie to jest konieczne dla odróżnienia we wzorcach nazw konstruktorów od nazw wprowadzanych stałych.

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

⟨wyrażenie⟩ ::= ⟨Identyfikator⟩ [⟨wyrażenie⟩]
⟨wzorzec⟩ ::= ⟨Identyfikator⟩ [⟨wzorzec⟩]

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

Przykład: Typy wariantowe:

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

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

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

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

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

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

Przykład:

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

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

Przykład: Zacyklone, nieskończone drzewa:

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

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

Przykład:

```
type 'a lista = Pusta | Pelna of 'a * 'a lista;;
type 'a lista = Pusta | Pelna of 'a * 'a lista

Pelna (4, Pusta);;
- : int lista = Pelna (4, Pusta)

Pelna ("ala", Pelna("ula", Pusta));;
- : string lista = Pelna ("ala", Pelna ("ula",
Pusta))

Pelna (4, Pelna ("ula", Pusta));;
error ...
```

4.9 Aliasy we wzorcach

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

⟨wzorzec⟩ ::= ⟨wzorzec⟩ as ⟨identyfikator⟩

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

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

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

4.10 Wyjątki

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

⟨jednostka kompilacji⟩ ::= ⟨deklaracja wyjątku⟩ | ...
⟨deklaracja wyjątku⟩ ::= exception ⟨wariant⟩

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

⟨wyrażenie⟩ ::= raise ⟨wyrażenie⟩

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

⟨wyrażenie⟩ ::= try ⟨wyrażenie⟩ with { ⟨wzorzec⟩ -> ⟨wyrażenie⟩ | }*
⟨wzorzec⟩ -> ⟨wyrażenie⟩

Przykład:

```
exception Dzielenie_przez_zero of float;;
exception Dzielenie_przez_zero of float

let dziel x y =
  if y = 0.0 then
    raise (Dzielenie_przez_zero x)
  else x /. y;;
val dziel : float -> float -> float = <fun>

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

odwrotnosc 42.;;
- : float = 0.0238095238095238082

odwrotnosc 0.0;;
- : float = 0.
```

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

```
let failwith s =
  raise (Failure s);;
```

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

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

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

Ćwiczenia

Ćwiczenie programistyczne na listy i inne struktury danych:

1. Lista n pierwszych liczb naturalnych.
2. Wyszukanie n-tego elementu listy.
3. Odwrócenie listy `rev`.
4. Sklejanie list `@`, `append`.
5. Zdublować elementy listy.
6. Napisz procedurę, która listę par postaci $[(n_1, x_1); (n_2, x_2); \ldots; (n_k, x_k)]$ przekształca w listę postaci $[\underbrace{x_1; \ldots; x_1}_{n_1 \text{ razy}}; \underbrace{x_2; \ldots; x_2}_{n_2 \text{ razy}}; \ldots; \underbrace{x_k; \ldots; x_k}_{n_k \text{ razy}}]$.
7. Napisz procedurę `last`, której wynikiem jest ostatni element listy.
8. Napisz procedurę `zsumuj`, która dla danej niemalejącej listy dodatnich liczb całkowitych $(a_1 \ldots a_n)$ oblicza listę $(b_1 \ldots b_n)$, gdzie $b_i = \sum_{k=a_i}^{n} a_k$. (Pamiętaj o tym, że jeśli $m > n$, $\sum_{k=m}^{n} a_k = 0$.)
9. Napisz procedury `head` i `tail`, które dla zadanej listy l i liczby całkowitej n zwracają pierwsze/ostatnie n elementów listy l. Jeśli lista l ma mniej elementów niż n, to wynikiem powinna być cała lista l. (Nazwy pochodzą od programów `head` i `tail` w Uniksie.) Co jest wynikiem `tail (head l n) 1`?
10. Napisz program wybierający max i min z listy i dokonujący jak najmniejszej liczby porównań.
11. Napisz program wybierający większość z listy większościowej; uzasadnij jego poprawność.
12. Napisz procedurę `max_diff : int list` $\rightarrow$ `int`, która dla niepustej listy $[x_1; \ldots; x_n]$ znajdzie maksymalną różnicę $x_j - x_i$ dla $1 \leq i < j \leq n$.
13. Napisz procedurę `podziel: int list` $\rightarrow$ `int list list`, która dla danej listy postaci $[a_1; a_2; \ldots; a_n]$, zawierającej permutację zbioru $\{1, 2, \ldots, n\}$ znajdzie jej podział na jak najliczniejszą listę list postaci:

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

    taką że:

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

    Przyjmujemy, że wynikiem dla listy pustej jest lista pusta.
    Przykład: `podziel` $[2; 3; 1; 6; 5; 4; 7; 9; 10; 11; 8] = [[2; 3; 1]; [6; 5; 4]; [7]; [9; 10; 11; 8]]$.

14. Napisz procedurę `mieszaj`, która wywołana dla danej listy, obliczy listę powstałą przez następujące przestawienie elementów: na pozycjach nieparzystych mają kolejno znajdować się $\lceil \frac{n}{2} \rceil$ początkowe elementy danej listy, a na pozycjach parzystych mają znajdować się pozostałe elementy danej listy, w odwrotnej kolejności. Na przykład: `mieszaj [1;2;3;4;5] = [1;5;2;4;3]`

15. Napisz procedurę `trójki: int list → (int * int * int) list`, która dla zadanej listy dodatnich liczb całkowitych, uporządkowanej rosnąco, stworzy listę takich trójek (a, b, c) liczb z danej listy, że:

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

16. Zdefiniuj taką procedurę `przedziały: int list → int*int`, że `przedziały` $[a_1; \ldots; a_n] = (k, l)$, gdzie jest to taka para liczb $1 \leq k \leq l \leq n$, dla której suma $a_k + \cdots + a_l$ jest największa.

17. Załóżmy, że dana jest lista $[x_1; x_2; \ldots; x_n]$. *Sufiksem* tej listy nazwiemy każdą listę, którą można uzyskać przez usunięcie pewnej liczby (od 0 do n) jej początkowych elementów. Tak więc sufiksami danej listy będzie n.p. ona sama, pusta lista, a także $[x_3; x_4; \ldots; x_n]$.

    Napisz procedurę `tails` : α `list` → α `list list`, która dla danej listy tworzy listę wszystkich jej sufiksów, uporządkowaną wg malejących ich długości.

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

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

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

    ```
    dekompresuj [1; 3; 3; 9; 42; 3];;
    - :  int list = [1; 2; 2; 5; 11; 11; 2]
    ```

19. Palindrom, to taka lista p, że $p =$ `rev` p. Napisz procedurę `palindrom` : α `list` → `int`, która dla danej listy obliczy długość jej najdłuższego spójnego fragmentu, który jest palindromem.

20. Napisz procedurę `podziel : int -> 'a list -> 'a list list`, która dla $k > 0$ i listy $[x_1; \ldots; x_n]$ podzieli ją na listę list postaci:

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

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

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

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

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

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

    Uwaga: Pusta lista reprezentuje zerowy wielomian.

23. Zdefiniuj typ reprezentujący drzewa o wierzchołkach dowolnego (skończonego stopnia). Zdefiniuj garść procedur operujących na takich drzewach (np. głębokość, liczbę elementów, lista elementów w porządku prefiksowym/postfiksowym).

24. Rozważmy drzewo binarne, w którego wierzchołkach pamiętane są różne dodatnie liczby całkowite. Dane są dwie listy: wyniki obejścia drzewa w porządku infiksowym i prefiksowym. Napisz procedurę, która odtwarza drzewo na podstawie takich dwóch list.

25. Napisz procedurę, która dla dowolnego drzewa binarnego poprawia je tak, że spełnia ono słabszą wersję warunku BST: dla dowolnego węzła, lewy syn nie jest większy, a prawy nie jest mniejszy, niż węzeł.

26. Napisz procedurę `rozcięcie`, która znajduje taką krawędź w danym drzewie binarnym, że po jej usunięciu drzewo rozpada się na dwa drzewa o minimalnej różnicy liczb węzłów. (Wynikiem procedury powinien być dolny koniec szukanej krawędzi.)

27. Dana jest deklaracja typu drzew binarnych:

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

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

28. Dany jest typ reprezentujący drzewa binarne:

```
type α drzewo =
  Drzewo of α drzewo * α * α drzewo |
  Lisc;;
```

Zdefiniuj typ α `przechadzka` oraz procedury:

```
buduj:  α drzewo → α przechadzka
w_lewo:  α przechadzka → α przechadzka
w_prawo:  α przechadzka → α przechadzka
w_gore:  α przechadzka → α przechadzka
obejrzyj:  α przechadzka → α
```

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

Podczas przechadzki po drzewie możemy przebywać jedynie w wierzchołkach `Drzewo _`. Jeśli nastąpi próba wejścia do liścia `Lisc` lub wyjścia „ponad" korzeń, należy podnieść wyjątek `Poza_drzewem`.

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

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

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

Wytyczne dla prowadzących ćwiczenia

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

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

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

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

Ad. 19 Oczekujemy rozwiązania o złożoności czasowej $\Theta(n^2)$.

Wykład 5. Budowanie abstrakcji za pomocą danych

Tworząc strukturę danych musimy podjąć kilka decyzji projektowych:

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

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

Operacje tworzące interfejs możemy podzielić na trzy kategorie:

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

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

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

programy korzystające ze złożonej struktury danych	
	modyfikatory
implementacja modyfikatorów	
	konstruktory i selektory
implementacja struktury danych	
	prostsze typy i operacje na nich
realizacja typów w języku programowania i wykorzystywane struktury danych	

5.1 Przyład: Liczby wymierne

5.1.1 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 q1 q2`.

Wymagamy przy tym, aby zachodziły następujące równości:

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

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

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

5.1.2 Modyfikatory

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

$$\frac{l_1}{m_1} + \frac{l_2}{m_2} = \frac{l_1m_2 + l_2m_1}{m_1m_2}$$

$$\frac{l_1}{m_1} - \frac{l_2}{m_2} = \frac{l_1m_2 - l_2m_1}{m_1m_2}$$

$$\frac{l_1}{m_1} \cdot \frac{l_2}{m_2} = \frac{l_1l_2}{m_1m_2}$$

$$\frac{l_1/m_1}{l_2/m_2} = \frac{l_1m_2}{l_2m_1} \quad \text{dla} \quad l_2 \neq 0$$

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

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

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

let div_wym x y =
  ulamek
    (licznik x * mianownik y)
    (mianownik x * licznik y)
```

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

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

```
let equal_wym x y =
  licznik x * mianownik y = licznik y * mianownik x;;
```

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

```
let ulamek l m = (l, m);;
let licznik (l, m) = l;;
let mianownik (l, m) = m;;
```

5.1.4 Zmiana reprezentacji liczb wymiernych

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

```
let ulamek l m =
  let r = nwd l m
  in ((l / r), (m / r));;
let licznik (l, m) = l;;
let mianownik (l, m) = m;;
```

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.

5.2 Kiedy bariery abstrakcji są właściwe?

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

1. wymyślamy zestaw zmian, 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) trzeba by zmodyfikować,

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

5.3 Podsumowanie wprowadzonych zasad

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

- **Technika pobożnych życzeń** — rozwiązując problem top-down dzielimy go na 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.

Wykład 6. Procedury wyższych rzędów jako abstrakcje konstrukcji programistycznych

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

Wcześniej zaznaczyliśmy, że procedury są w funkcyjnych językach programowania obywatelami pierwszej kategorii, czyli są równie dobrymi wartościami jak wszystkie inne. W szczególności procedury mogą być argumentami procedur lub ich wynikami. Argumenty proceduralne występują również w imperatywnych językach programowania (np. w standardzie Pascala). Natomiast to, że procedury mogą być wynikami procedur, jest czymś charakterystycznym dla języków funkcyjnych[2].

6.1 Typy proceduralne i polimorfizm

Przyjrzyjmy się kilku prostym przykładom procedur wyższych rzędów.

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

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

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

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

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

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

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

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

podstawiając za `'a` typ `int`, uzyskujemy z typu `('a -> 'a) -> ('a -> 'a)` typ `(int -> int) -> (int -> int)`. Natomiast podstawiając za `'a` typ `'a -> 'a`, uzyskujemy typ `(('a -> 'a) -> ('a -> 'a)) -> (('a -> 'a) -> ('a -> 'a))`.

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

Przykład: Spójrzmy na następujące zastosowanie `twice`:

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

Procedura `czterokrotnie` działa w następujący sposób:

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

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

Składanie procedur możemy zdefiniować następująco:

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

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

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

Procedurę `twice` możemy teraz zdefiniować w następujący sposób:

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

Możemy też zdefiniować wielokrotne składanie funkcji:

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

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

6.2 Czy istnieją procedury wieloargumentowe?

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

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

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

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

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

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

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

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

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

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

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

```
let twice f x = f (f x);;
let compose f g x = f (g x);;
let curry f x y = f (x, y);;
let uncurry f (x, y) = f x y;;
```

6.3 Sumy częściowe szeregów

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

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

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

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

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

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

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

6.4 Różniczkowanie funkcji

Zdefiniujmy najpierw różniczkę:

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

Wówczas pochodną możemy przybliżyć następująco:

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

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

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

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

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

6.5 Szukanie zer

Przedstawimy dwie metody szukania zer funkcji:

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

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

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

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

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

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

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

6.5.1 Metoda przez bisekcję

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

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

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

Wykorzystujemy procedurę `iter`.

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

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

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

and popraw x =
  let l = fst x
  and p = snd x
  in
    let srodek = average l p
    in
      if dodatnie (f srodek) then
        (l, srodek)
      else
        (srodek, p)
```

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

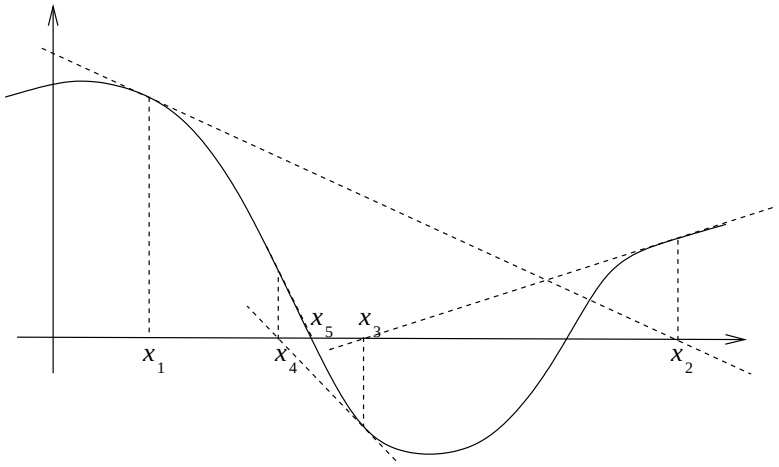
Oto przykład zastosowania szukania zer przez bisekcję do pierwiastkowania liczb:

```
let sqrt a =
  let f x = a -. square x
  in bisekcja f 0.0 (a +. 1.0);;
```

```
val sqrt : float -> float = <fun>

sqrt 42.0;;
- : float = 6.48074069840786
```

6.6 Metoda Newtona

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



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

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

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

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

Pod nazwą „metoda Newtona" znana jest też metoda liczenia pierwiastków kwadratowych. Jest to szczególny przypadek metody Newtona znajdowania zer, dla funkcji $f(x) = x^2 - a$.

Funkcja ta ma zera w punktach $\pm\sqrt{a}$. Ponadto $f'(x) = 2x$, czyli nasz algorytm przekształca przybliżenie x w:

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

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

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

sqrt 17.64;;
- : float = 4.20000000000023643
```

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

6.7 Punkty stałe funkcji

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

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

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

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

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

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

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

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

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

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

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

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

6.8 Procedury wyższych rzędów i listy

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

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

6.8.1 `fold_left`

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

W schemacie tym mamy następujące elementy zmienne:

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

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

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

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

Oto kilka prostych przykładów.

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

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

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

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

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

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

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

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

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

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

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

6.8.2 `fold_right`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Zwróćmy uwagę, że wartość kumulowana przez `fold_right` to procedura. Procedura ta przekształca wynik skumulowany przez `fold_left` na podstawie elementów listy, które nie zostały jeszcze przejrzane przez `fold_right`, w wynik ostateczny.

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

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

6.8.3 `map`

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

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

Przykład: Przykłady użycia `map`:

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

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

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

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

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

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

6.8.4 `filter`

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

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

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

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

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

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

let eratostenes n =
  let rec erat l = if l = [] then [] else (List.hd l)::(erat (sito l))
  in erat (gen n);;
val eratostenes : int -> int list = <fun>

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

6.9 Zastosowania procedur wyższych rzędów

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

Zastosowania procedur wyższych rzędów można podzielić na cztery grupy:

- Pewne pojęcia matematyczne, zwłaszcza te dotyczące funkcji, w naturalny sposób przekładają się na procedury wyższych rzędów, np.: sumy częściowe szeregów, składanie funkcji, różniczkowanie funkcji itp.

- Procedury są jeszcze jednym typem danych. Przyjmując takie podejście, procedury przetwarzające dane, jeśli te dane będą akurat procedurami, same będą procedurami wyższych rzędów.

- Procedury wyższych rzędów są również narzędziem abstrakcji. Jeżeli ten sam fragment kodu pojawia się w kilku miejscach, to naturalnym jest wyłonienie go w postaci (zwykłej) procedury. Jeżeli ten sam schemat kodu pojawia się w wielu miejscach, ale różni się wypełniającymi go fragmentami, to schemat ten możemy ująć w postaci procedury wyższego rzędu, a fragmenty do wypełnienia staną się parametrami proceduralnymi. Przykładem może być tu procedura `iteruj`.

- Zastosowanie standardowych procedur wyższych rzędów przetwarzających listy daje szczególnie zwięzłe definicje operacji na listach.

Laboratorium

Zabawy z procedurami wyższych rzędów:

- przećwicz składanie prostych funkcji,
- uniwersalne potęgowanie (dostarczamy element neutralny i operację mnożenia, a wynikiem jest podnoszenie do potęgi całkowitej); zinstancjonuj uniwersalne potęgowanie tak, żeby otrzymać: potęgowanie liczb całkowitych, potęgowanie liczb zmiennopozycyjnych, podnoszenie funkcji do potęgi (inne?).
- różniczkowanie i całkowanie funkcji,
- 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. Wyznacz eksperymentalnie, ile razy należy stosować tłumienie w zależności od n. (Podpowiedź: k-krotne uśrednienie funkcji f, to funkcja postaci: `fun` $x \to \frac{x \cdot (2^k - 1) + f(x)}{2^k}$.)
- Zaimplementuj pierwiastkowanie, jako instancję szukania zer funkcji.
- Zadanie o Origami, czas 2 tygodnie, wartość, 3*.

  Dana jest lista prostych zorientowanych wyznaczająca ciąg złożeń papieru. Kartka papieru to kwadrat jednostkowy. Każda prosta jest reprezentowana przez parę różnych punktów. Punkt to para współrzędnych x i y. Papier jest składany w ten sposób, że z prawej strony prostej (patrząc w kierunku od pierwszego punktu do drugiego) jest przekładany na lewą. Napisz procedurę `origami`, która dla wywołania `origami l p` oblicza ile warstw papieru znajdzie się w punkcie `p` po złożeniu papieru zgodnie z prostymi z listy `l` (Przyjmujemy, że na linii złożenia są obie składane warstwy papieru.)

  Rozwiązując to zadanie należy wykorzystać funkcyjny charakter języka.

Ćwiczenia

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

   ```
   iterate 2 (function x -> x * (x+1)) 2
   iterate 3 (function x -> x * (x+1)) 1
   ```

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

2. Niech $f : \mathcal{R} \to \mathcal{R}$ będzie funkcją 1–1 i „na” oraz taką, że $f(0) = 0$, f jest rosnąca i $|f(x)| \geq |x|$. Zaimplementuj procedurę `odwrotność`, której wynikiem dla parametru f będzie przybliżenie f^{-1} z dokładnością zadaną przez stałą `epsilon` (czyli jeśli $g =$ (`odwrotność` f), to $\forall x \; |g(x) - f^{-1}(x)| \leq$ `epsilon`).

3. Wygładzenie funkcji z odstępem dx polega na uśrednieniu $f(x - dx)$, $f(x)$ i $f(x + dx)$. Napisz procedurę wygładzającą daną funkcję z zadanym odstępem.

4. Zaimplementuj aproksymację funkcji za pomocą szeregu Taylora. Twoja procedura powinna mieć następujące parametry: liczbę sumowanych wyrazów szeregu, punkt, w którym badana jest przybliżana funkcja. Wynikiem powinno być przybliżenie funkcji. Zastosuj przybliżenie pochodnej oraz sumy częściowe szeregów, przedstawione na wykładzie.

Wykład 7. Model obliczeń

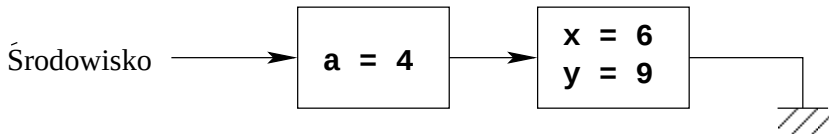
W tym wykładzie przedstawimy uproszczoną semantykę operacyjną poznanego fragmentu Ocamla. Z jednej strony będzie to model wykonywania obliczeń na tyle dokładny, że pozwoli nam określić wyniki, a później również złożoność czasową i pamięciową obliczeń. Z drugiej strony będzie on uproszczony, gdyż pewne kwestie pominiemy, np. kontrolę typów, rozmaite optymalizacje wykonywane przez kompilator, czy sposób realizacji tych elementów języka, których jeszcze nie poznaliśmy. Wyjaśnimy też pewne szczegóły wykonywania obliczeń, które pominęliśmy wcześniej.

7.1 Środowisko i ramki.

Przedstawiając podstawy Ocamla mówiliśmy o *środowisku* przypisującym nazwom stałe ich wartości. Dodatkowo takich środowisk może być wiele: środowisko zawierające wszystkie aktualnie globalnie zdefiniowane stałe, środowiska powstające na potrzeby definicji lokalnych i wywołań procedur.

Środowisko jest zrealizowane jako lista jednokierunkowa, tzw. *lista ramek*. Każda ramka zawiera identyfikator (lub identyfikatory) i ich wartości (lub wskaźniki do takich wartości) oraz wskaźnik do kolejnej ramki tworzącej środowisko. Poszukując w środowisku wartości stałej, przeglądamy kolejne ramki środowiska. Pierwsza ramka, która zawiera nazwę stałej, określa jej wartość.

Dalej będziemy utożsamiać środowisko ze wskaźnikiem do pierwszej ramki tworzącej środowisko. Środowisko reprezentowane przez listę ramek bez pierwszej ramki będziemy nazywać *środowiskiem otaczającym*, a wskaźnik prowadzący z jednej ramki do kolejnej będziemy nazywać *wskaźnikiem do środowiska otaczającego*.

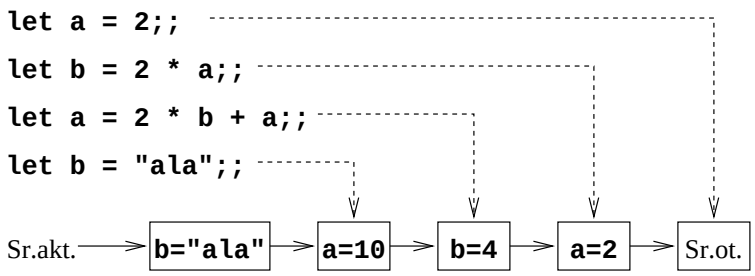


7.2 Definicje globalne

Prowadząc interakcję z kompilatorem, pracujemy w kontekście tzw. *aktualnego środowiska (globalnego)*. Każda dodana definicja zmienia to środowisko. Zdefiniowanie nowych wartości powoduje dodanie nowej ramki zawierającej zdefiniowane wartości. Dotychczasowe środowisko staje się środowiskiem otaczającym tej ramki. Utworzona ramka staje się nowym aktualnym środowiskiem (globalnym).

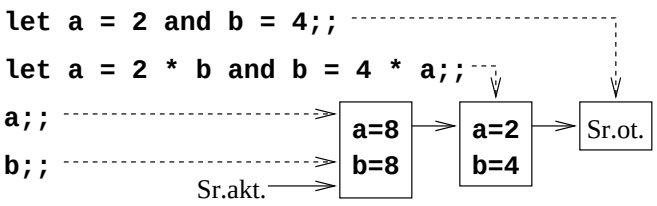
Mechanizm ten pozwala na przesłanianie istniejących w środowisku stałych. Jeśli zdefiniujemy nową stałą o takiej samej nazwie, to będzie ona znajdowała się we wcześniejszej ramce i to właśnie ona będzie znajdowana w aktualnym środowisku. Jednak poprzednio zdefiniowana stała nie jest usuwana ze środowiska. Jak zobaczymy, będzie to miało znaczenie przy obliczaniu procedur odwołujących się do wcześniej zdefiniowanych stałych.

Przykład:



Ramka może zawierać więcej niż jedną stałą, jeżeli w definicji użyjemy spójnika `and`.

Przykład:



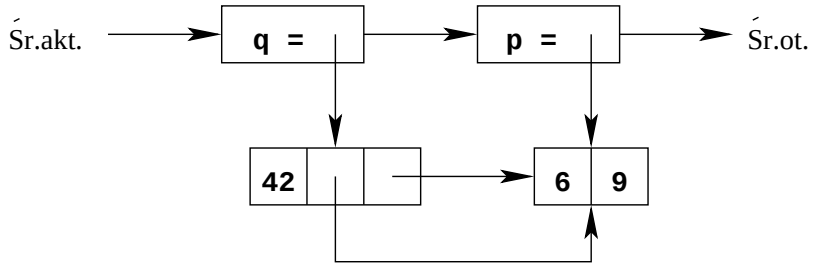
7.3 Wartości typów danych

Wartości prostych typów wbudowanych (takie jak `int`, `bool`, czy `float`) są pamiętane w ramkach wprost. Wartości typów złożonych są pamiętane jako wskaźniki do odpowiednich struktur wskaźnikowych reprezentujących te wartości. Dzięki przyjęciu takiej konwencji, reprezentacja dowolnej wartości zajmuje zawsze w ramce stałą pamięć. (Dotyczy to zarówno wartości przechowywanych w ramkach, jak i przekazywanych obliczanych wartości wyrażeń.) W rezultacie definiując stałe o złożonych wartościach nie kopiujemy tych wartości, a jedynie wskaźniki do nich. Dodatkowo, ponieważ raz powstałe wartości nie ulegają zmianie, struktury danych reprezentujące różne wartości mogą współdzielić pewne fragmenty.

Wartości różnych złożonych typów danych są generalnie reprezentowane jako rekordy złożone ze wskaźników do wartości składowych:

- Element produktu kartezjańskiego jest reprezentowany jako n-tka wskaźników do tworzących go wartości współrzędnych.
- Rekord — podobnie jak produkt kartezjański jest reprezentowany jako rekordy złożony z wskaźników do wartości pól.
- Warianty bez argumentów są pamiętane jako stałe wyznaczające operator, a warianty z argumentami są pamiętane jako pary: stała wyznaczająca operator i wskaźnik do argumentu.
- Lista pusta jest reprezentowana jako stała, a lista niepusta jest reprezentowana jako para: głowa i ogon.

```
let p = (6,9);;
let q = (42, p, p);;
```



7.4 Wartości proceduralne

Na wartość procedury składają się trzy elementy:

- nazwy parametrów formalnych procedury,
- treść procedury i
- wskaźnik do środowiska, w którym zdefiniowano procedurę.

Pierwsze dwa elementy przekładają się w trakcie kompilacji na kod procedury. Natomiast wskaźnik do środowiska, w którym zdefiniowano procedurę, może być ustalony dopiero w trakcie obliczeń — dotyczy to np. procedur lokalnych. Reasumując, wartość procedury jest reprezentowana jako para wskaźników: do kodu skompilowanej procedury i do środowiska, w którym procedura została zdefiniowana. W przypadku procedur rekurencyjnych „środowisko, w którym zdefiniowano procedurę" zawiera jej własną definicję.

Uwaga: Formalnie rzecz biorąc, każda procedura ma tylko jeden argument. Jednak w sytuacjach, gdy takie uproszczenie nie będzie prowadzić do nieporozumień, będziemy dopuszczać obiekty proceduralne z wieloma argumentami. Uproszczenie takie jest możliwe wtedy, gdy w zastosowaniach procedury wszystkie jej parametry otrzymują wartości.

7.4.1 Zastosowanie procedury do argumentów

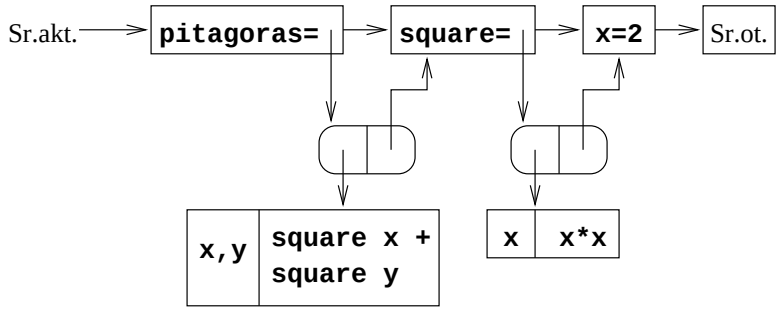
Zastosowanie procedury, czyli obliczenie jej wartości, polega na:

- wyliczeniu wartości wszystkich potrzebnych elementów: procedury i jej argumentów,
- stworzeniu nowej ramki, dla której środowiskiem otaczającym jest środowisko, w którym zdefiniowano procedurę, a nazwom parametrów formalnych przyporządkowano wartości argumentów,
- wyliczeniu w takim środowisku wartości wyrażenia stanowiącego treść procedury.

Przykład: Wprowadzenie definicji:

```
let x = 2;;
let square x = x * x;;
let pitagoras x y = square x + square y;;
```

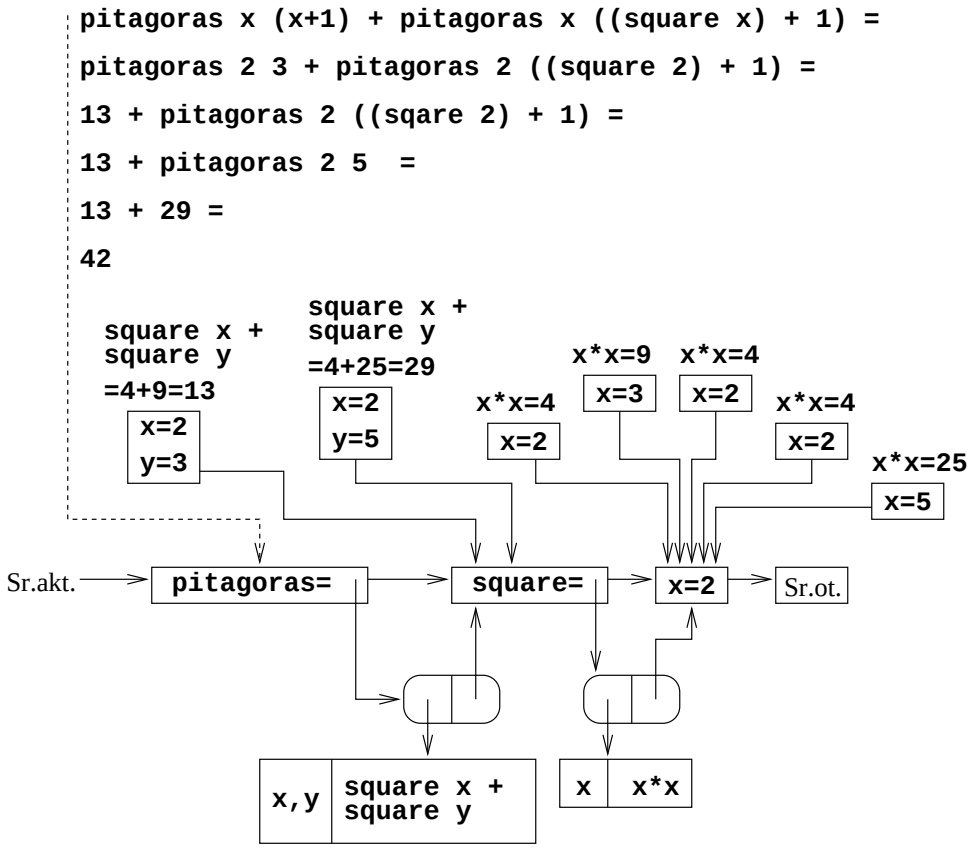
powoduje następujące rozszerzenie środowiska:



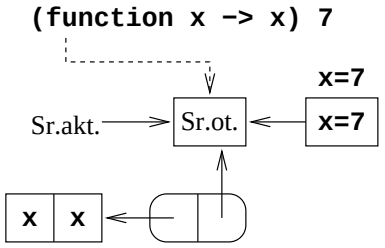
Obliczenie wyrażenia:

```
pitagoras x (x + 1) + pitagoras x ((square x) + 1);;
```

będzie przebiegać następująco:

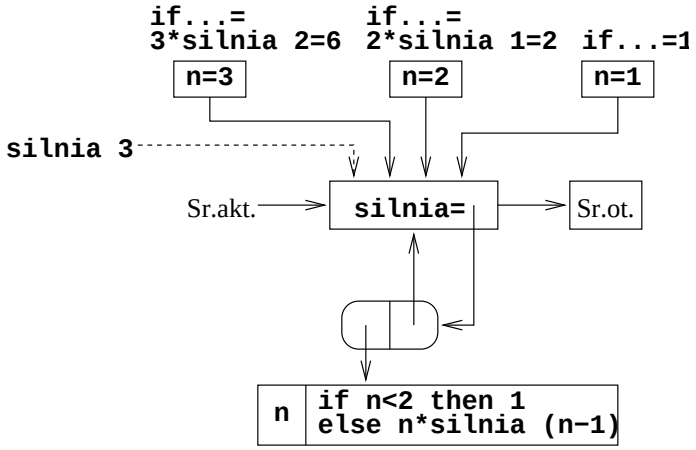


Przykład: Prześledźmy obliczenie prostego wyrażenia zawierającego λ-abstrakcję:



Przykład: Zastosowanie procedury rekurencyjnej:

```
let rec silnia n =
  if n < 2 then 1 else n * silnia (n - 1);;
silnia 3;;
```

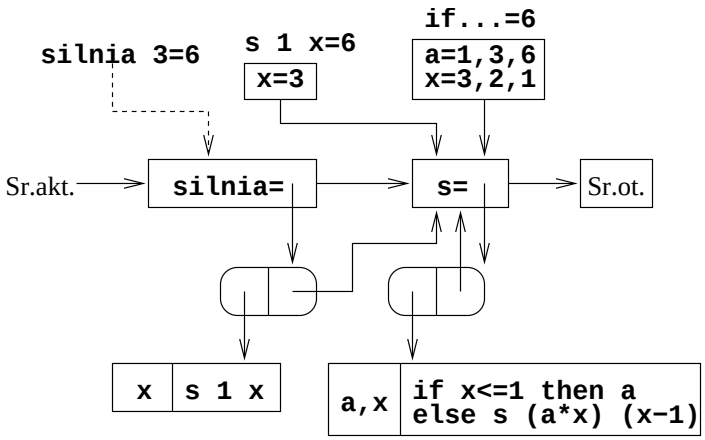


7.4.2 Rekurencja ogonowa

Jeśli wartości zwracane przez wszystkie wywołania rekurencyjne, bez żadnego przetwarzania, są przekazywane jako wynik danej procedury, to mówimy o rekurencji ogonowej. W przypadku rekurencji ogonowej, w momencie wywołania rekurencyjnego nie musimy tworzyć nowej ramki — możemy wykorzystać istniejącą ramkę, zmieniając jedynie pamiętane w niej wartości argumentów. W przypadku rekurencji ogonowej koszt pamięciowy związany z ramkami dla kolejnych wywołań rekurencyjnych jest stały, gdyż jest to tylko jedna ramka.

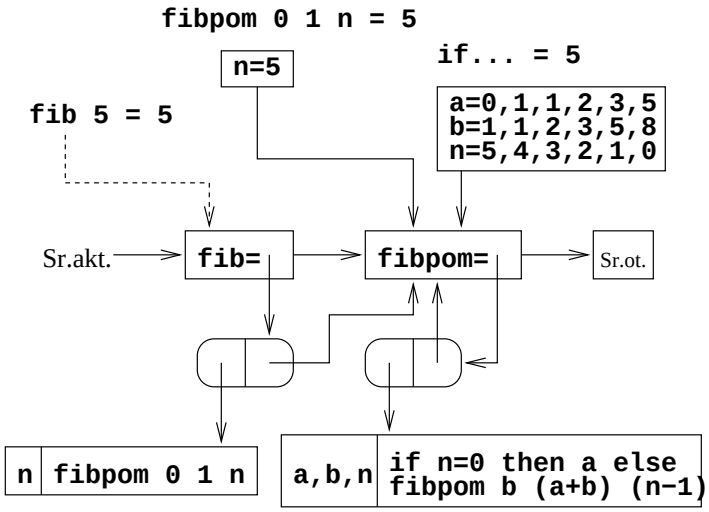
Przykład: Oto procedura obliczająca silnię z rekurencją ogonową. Zwróćmy uwagę na dodatkowy parametr `a`, w którym kumuluje się wynik. Takie dodatkowe parametry są nazywane *akumulatorami*. Dzięki zastosowaniu akumulatora mamy rekurencję ogonową.

```
let rec s a x =
  if x <= 1 then a else s (a * x) (x - 1);;
let silnia x = s 1 x;;
silnia 3;;
```



Przykład: Oto procedura obliczająca liczby Fibonacciego z rekurencją ogonową i akumulatorem.

```
let rec fibpom a b n =
  if n = 0 then a else fibpom b (a + b) (n - 1);;
let fib n = fibpom 0 1 n;;
fib 5;;
```



7.4.3 Definicje lokalne

Mówiliśmy wcześniej, że wyrażenie postaci:

`let` a `=` b `in` c

jest podobne do zastosowania λ-abstrakcji postaci:

`(function` a `->` c`)` b

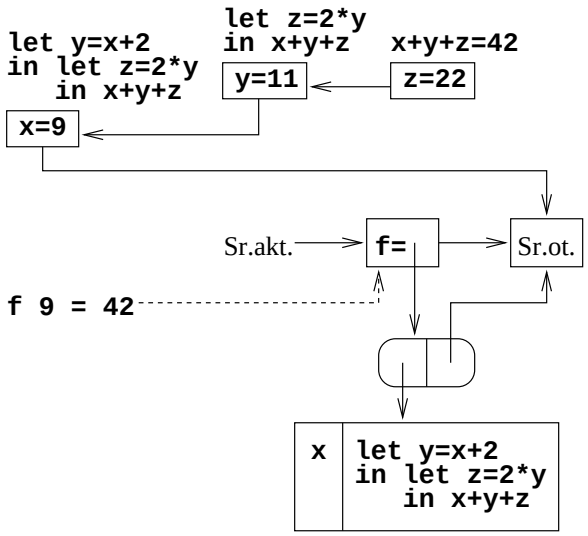
Sposób obliczania wyrażenia `let ...in ...` jest analogiczny do sposobu obliczania takiej λ-abstrakcji:

- tworzona jest ramka zawierająca definicję symboli lokalnych (otaczającym ją środowiskiem jest aktualne środowisko),
- w tak powstałym środowisku wyliczana jest wartość wyrażenia stojącego po `in`,
- po czym przywracane jest aktualne środowisko sprzed obliczenia całego wyrażenia.

Jeżeli definicje lokalne są pozagnieżdżane, to powstaje odpowiednio więcej ramek z wartościami symboli lokalnych.

Przykład: Następujący przykład pokazuje sposób realizacji definicji lokalnych.

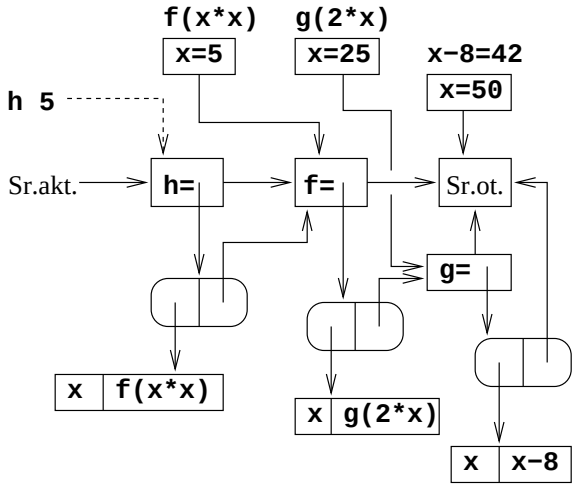
```
let f x =
  let y = x + 2
  in
    let z = 2 * y
    in
      x + y + z;;
f 9;;
```



Przykład: Niniejszy przykład ilustruje mniej typową kombinację użycia definicji lokalnych i procedur.

```
let f =
  let g x = x - 8
  in
    fun x -> g (2 * x);;
let h x = f (x * x);;
h 5;;
```

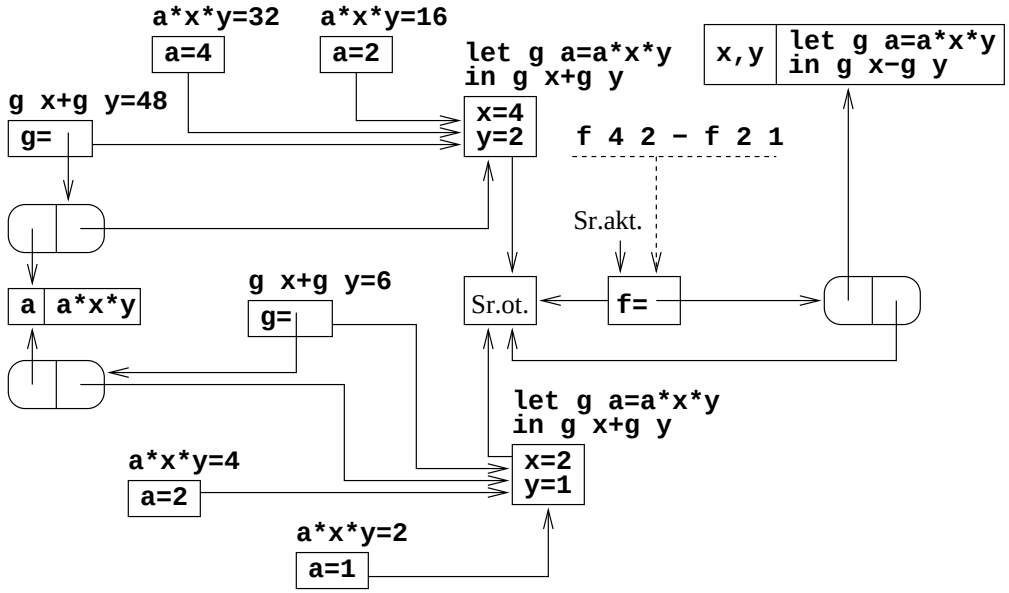
Zwróćmy uwagę na procedurę pomocniczą g. Procedura ta jest widoczna tylko wewnątrz procedury f, a jednak jej definicja ma tylko jedną instancję.



Przykład:

```
let f x y =
  let
    g a = a * x * y
  in
    g x + g y;;
f 4 2 - f 2 1;;
```

W tym przykładzie, w każdym wywołaniu procedury f powstaje osobna instancja procedury lokalnej g. Jest to naturalne zważywszy, że procedura g korzysta z argumentów wywołania procedury f.



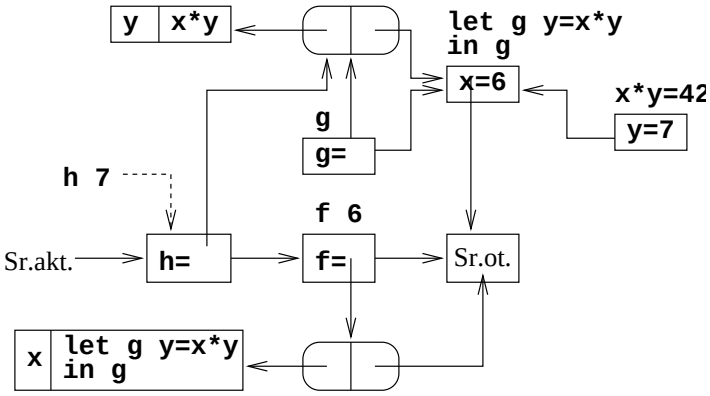
7.5 Ramki a rekordy aktywacji

Ramkom odpowiadają w językach imperatywnych *rekordy aktywacji*. Rekord aktywacji to porcja informacji towarzysząca pojedynczemu wywołaniu procedury. W rekordach aktywacji pamiętane są np. wartości zmiennych lokalnych. Rekordy aktywacji są pamiętane na stosie. W momencie powrotu z wywołania procedury, związany z nim rekord aktywacji jest niszczony.

Pokażemy na przykładach, że w naszym modelu obliczeniowym tak nie jest. Wynika to z funkcyjnego charakteru języka oraz z tego, że procedury mogą być wynikami procedur. Ramka, zawierająca lokalne definicje lub argumenty procedury, nie musi stawać się od razu zbędna, gdyż mogą na nią wskazywać wartości proceduralne. Jeśli spojrzymy na wszystkie ramki, to nie tworzą one listy ani stosu, ale drzewo (wskaźniki na otaczające środowisko prowadzą w górę tego drzewa). Ramki, które trzeba trzymać w pamięci, bo są potrzebne też nie muszą tworzyć listy, tylko drzewo. Zjawisko to występuje np. wówczas, gdy wynikiem procedury jest procedura mająca dostęp do lokalnie zdefiniowanych symboli.

Przykład: Niniejszy przykład pokazuje sytuację, gdy ramki powstałe w czasie wywołania procedury mogą być potrzebne po jego zakończeniu.

```
let f x =
  let g y = x * y
  in g;;
let h = f 6;;
h 7;;
```

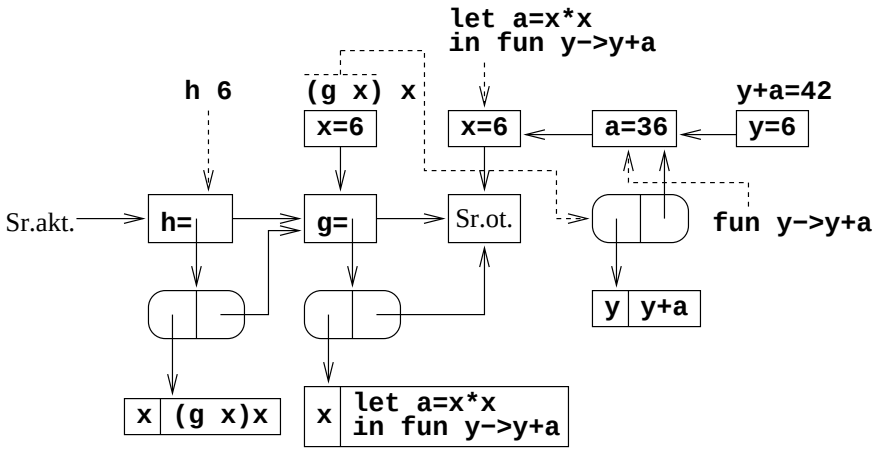


Ramka zawierająca definicję procedury `g` powstaje w wyniku wywołania `f 6`, ale jest potrzebna później, do obliczenia `h 7`.

Przykład: Poniższy przykład ilustruje sposób przekazywania wielu argumentów, tak jak to jest realizowane — po jednym na raz.

```
let g x =
  let
    a = x * x
  in
    fun y -> y + a;;
```

```
let h x = (g x) x;;
h 6;;
```



7.6 Odśmiecanie

W przedstawionym modelu obliczeń cały czas są tworzone ramki i elementy struktur danych. Co więcej, jak widzieliśmy, nie możemy usuwać ramek powstałych na potrzeby definicji lokalnych i zastosować procedur po obliczeniu wyników, gdyż mogą one być nadal potrzebne. Gdy brakuje wolnej pamięci, uruchamiany jest proces *odśmiecania*. Proces ten usuwa wszystkie te ramki i elementy struktur danych, które nie są dostępne i tym samym nie mogą być już do niczego potrzebne.

W tym celu przeszukuje się strukturę wskaźnikową ramek i danych. Za punkt wyjścia służą wszystkie te środowiska, w kontekście których są aktualnie obliczane wyrażenia, plus aktualne środowisko globalne. Wszystkie te ramki i rekordy, które są z nich dostępne (pośrednio lub bezpośrednio) są potrzebne. Natomiast te, które są niedostępne, mogą być usunięte.

Obliczając złożoność czasową programów możemy pominąć koszt odśmiecania. Zwykle odśmiecanie jest tak zrealizowane, że jego koszt zamortyzowany jest stały. Można go wliczyć w koszt czasowy tworzenia nowych ramek i rekordów, obciążając utworzenie każdej ramki i rekordu stałym kosztem czasowym.

Obliczanie kosztu pamięciowego jest skomplikowane. Należy prześledzić plątaninę powstających ramek i rekordów, sprawdzić które są w danym momencie niezbędne, a które mogą zostać odśmiecone i określić minimalną wielkość pamięci niezbędnej do przeprowadzenia obliczeń. Dodatkowo możemy przyjąć konwencję, że rozmiar danych nie wlicza się do złożoności pamięciowej. Wynika to z charakteru programowania funkcyjnego. Przekazane dane nie mogą być modyfikowane przez program.

Ćwiczenia

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

1. Napisz procedurę `exists`, która dla danego predykatu i listy sprawdzi, czy na liście jest element spełniający predykat. Wykorzystaj wyjątki tak, aby nie przeglądać listy, gdy to już nie jest potrzebne.

2. Napisz procedurę negującą predykat `non: ('a -> bool) -> ('a -> bool)`. Za pomocą tej procedury oraz procedury `exists` zdefiniuj procedurę `forall`, która sprawdza, czy dany predykat jest spełniony przez wszystkie elementy danej listy. Czy zastosowanie wyjątków w implementacji procedury `exists` nadal powoduje, że przeglądane są tylko niezbędne elementy listy?

3. Zapisz procedurę `append` za pomocą `fold_right/fold_left`.

4. Napisz procedurę obliczającą funkcję będącą sumą listy funkcji.

5. Napisz procedurę obliczającą funkcję będącą złożeniem listy funkcji.

6. Zapisz za pomocą `map` i `flatten` procedurę `heads`, której wynikiem dla danej listy list, jest lista pierwszych elementów niepustych list składowych. Puste listy składowe nie powinny wpływać na wynik.

7. Napisz procedurę obliczającą sumę elementów listy występujących po ostatniej liczbie ujemnej (lub wszystkich, jeżeli na liście nie ma liczb ujemnych).

8. Napisz procedurę `sumy : int list -> int list`, która dla danej listy $[x_1, \ldots, x_n]$ oblicza listę postaci: $[x_1, x_1 + x_2, x_1 + x_2 + x_3, \ldots, x_1 + x_2 + \cdots + x_n]$.

   Przykład: `sumy [1; 5; 2; 7; 12; 10; 5] = [1; 6; 8; 15; 27; 37; 42]`.

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

10. Oblicz ciąg różnicowy zadanej listy liczb całkowitych.

11. Oblicz listę złożoną z pierwszych elementów kolejnych ciągów różnicowych danej listy.

12. Dany jest ciąg nawiasów, otwierających i zamykających. Napisz procedurę `nawiasy`, która obliczy minimalną liczbę nawiasów które należy obrócić, tak aby uzyskać poprawne wyrażenie nawiasowe. Jeżeli nie jest to możliwe, to należy podnieść wyjątek `NieDaSię`.

    ```
    exception NieDaSię
    type nawias = Otwierający | Zamykający
    val nawiasy : nawias list → int
    ```

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

    Napisz procedurę `usrednienie`, która dla danej listy obliczy jej uśrednienie.

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

15. Dana jest lista $[x_1; \ldots; x_n]$ i stała $0 \leq k < n$. Oblicz listę $[y_1; \ldots; y_n]$, gdzie $y_i = \min(x_{\max(i-k,1)}, \ldots, x_i)$.

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

17. Załóżmy, że dana jest lista $[x_1; x_2; \ldots; x_n]$. *Sufiksem* tej listy nazwiemy każdą listę, którą można uzyskać przez usunięcie pewnej liczby (od 0 do n) jej początkowych elementów. Tak więc sufiksami danej listy będzie n.p. ona sama, pusta lista, a także $[x_3; x_4; \ldots; x_n]$.

    Napisz (za pomocą `fold_left`/`fold_right`) procedurę `tails` : α `list` $\rightarrow \alpha$ `list list`, która dla danej listy tworzy listę wszystkich jej sufiksów, uporządkowaną wg malejących ich długości.

18. Napisz procedurę `pojedyncze` : α `list` $\rightarrow \alpha$ `list`, która dla danej listy wybierze z niej te elementy, które występują dokładnie raz z rzędu, np.:

    $$\texttt{pojedyncze}\ [1; 1; 3; 5; 5; 5; 3; 5] = [3; 3; 5]$$

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

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

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

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

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

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

    ```
    kompresuj [1; 2; 2; 5; 11; 11; 2];;
    - :  int list = [1; 6; 9; 42; 3]
    ```

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

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

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

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

$$\texttt{buduj\_permutacje}\ [c_0; \ldots; c_k]\ x = x$$

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

```
let p = buduj_permutacje [[2; 1]; [8; 3; 4]; [5; 7; 6; 10]; [11]];;
map p [1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12];;
-: int list = [2; 1; 4; 8; 7; 10; 6; 3; 9; 5; 11; 12]
```

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

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

    Przykład:

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

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

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

    Przykład:

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

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

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

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

```
type 'a tree = Node of 'a tree * 'a * 'a tree | Leaf;;
let rec fold_tree f a t =
  match t with
    Leaf -> a |
    Node (l, x, r) -> f x (fold_tree f a l) (fold_tree f a r);;
```

Użyj procedury `fold_tree` do zaimplementowania:

(a) Policzenia liczby węzłów w drzewie.

(b) Policzenia wysokości drzewa.

(c) Policzenia średnicy drzewa.

(d) Sprawdzenia czy drzewo jest drzewem BST (dla drzewa liczb `float`).

(e) Zaimplementowania procedury `map_tree` — odpowiednika procedury `map` dla drzew.

(f) Powiemy, że wartość w węźle drzewa jest *widoczna*, jeżeli na ścieżce od tego węzła do korzenia drzewa nie ma większej wartości (w sensie standardowego porządku $>$). W szczególności liczba w korzeniu drzewa jest zawsze widoczna, a liczby mniejsze od niej nie są nigdy widoczne.

Napisz procedurę `widoczne: int drzewo` $\rightarrow$ `int`, która dla zadanego drzewa (zawierającego wyłącznie nieujemne liczby całkowite) wyznaczy listę widocznych liczb.

(g) Skonstruowania listy elementów znajdujących się w wierzchołkach drzewa, w kolejności infiksowej.

26. Przuypomnij sobie definicję typu danych drzew dowolnego stopnia. Wymyśl i napisz odpowiedniki procedur `map`, `fold` i `filter` dla takich drzew.

Wytyczne dla prowadzących ćwiczenia

To duży zestaw zadań. Należy na niego poświęcić ok. 2 zajęć. Zamiast procedur rekurencyjnych należy stosować standardowe procedury wyższych rzędów przetwarzające listy. W zadaniach wymagających użycia `fold`-ów zwracamy uwagę, której procedury należy użyć: `fold_left`, `fold_right`, czy też nie ma to znaczenia. Dopóki na wykładzie nie będzie mowy o rzędach funkcji i analizie złożoności, nieformalnie analizujemy, czy rekurencja ogonowa w `fold_left` daje stały koszt pamięciowy, czy też i tak musi on być liniowy. Po omówieniu analizy złożoności na wykładzie analizujemy złożoność rozwiązań.

Nie wszystkie zadania zdąży się zrobić — pozostałe studenci mogą rozwiązać sami w ramach przygotowania do kolokwium. Warto jednak zrobić zadanie 25 dotyczące przetwarzania drzew.

Ad. 9 Należy zwrócić uwagę na przypadki brzegowe, w szczególności na to, czy rekurencja może się zapętlić.

Ad. 25 Należy zwrócić uwagę na złożoność czasową. Uzyskanie liniowej złożoności może wymagać kumulowania za pomocą `fold_tree` nie wyniku, ale odpowiedniej procedury konstruującej wynik.

Wykład 8. Analiza kosztów

8.1 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},
- $\log_2 n$,
- $2\sqrt{4n}$,
- n^n,
- $\ln(10n)$,
- 4^n,
- $n!$,
- 10^n,
- 0.

8.2 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.

8.3 Koszt czasowy

Liczba elementarnych operacji potrzebnych do wyliczenia zadanej wartości, jako funkcja [rozmiaru] danych. Co to są operacje elementarne:

- odczytanie wartości symbolu ze środowiska (stała, liczba) — 1,
- zastosowanie procedury do argumentów — koszt obliczenia wszystkich elementów (w tym procedury) + koszt wyliczenia treści procedury (w przypadku operatorów wbudowanych koszt ich obliczania wynosi 1, chyba że powiedziane jest inaczej) + liczba elementów kombinacji,
- `if` — koszt obliczenia warunku i zależnie od wyniku koszt obliczenia odpowiedniej części,
- wyrażenie `function` — 1 (uwaga: koszt ujawnia się dopiero przy zastosowaniu do argumentów),
- dopasowywanie wzorców `match-with` — łączna długość wzorców + koszt obliczenia wyrażenia odpowiadającego dopasowanemu wzorcowi.
- wyrażenie `let-in` — zgodnie z rozwinięciem do λ-wyrażenia i kombinacji, czyli koszt wyliczenia definiowanych lokalnie wartości + wyliczenie właściwego wyrażenia + 2,
- rekurencja — równanie rekurencyjne na kosztach.

Przykład:

- nierekurencyjny przykład z `let`:

```
let f =
  let g x = 3 * x
  in
    function x -> g (2 * x);;
f 7;;
```

- silnia,

8.4 Koszt pamięciowy

Liczymy pamięć zajmowaną przez ramki zawierające potrzebne symbole, pamięć zajmowaną przez tworzone wartości plus pamięć potrzebną do obliczania wyrażeń. Na potrzeby tego wykładu, nie wliczamy danych, natomiast wliczamy wartości pośrednie i wynik. Co to znaczy potrzebne symbole:

- wszystkie symbole widoczne w środowisku (nie przysłonięte) są potrzebne,
- jeśli wartością potrzebnego symbolu jest procedura, to symbole występujące w środowisku wskazywanym przez tę procedurę są potrzebne,
- jeśli potrzebna jest wartość złożona (np. lista, lub para), to potrzebne są również jej składowe.

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

- liczby, znaki i wartości logiczne mają rozmiar 1,
- n-tka ma rozmiar n,
- lista ma rozmiar taki jak długość + łączne rozmiary elementów,
- wartości algebraiczne — wielkość drzewa,
- 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.

8.5 Przykład: Potęgowanie

Możemy skorzystać ze wzorów:

$$\begin{aligned} b^n &= b \cdot b^{n-1} \\ b^0 &= 1 \end{aligned}$$

Co możemy zaimplementować od razu jako:

```
let rec potega b n =
  if n = 0 then 1 else b * potega b (n-1);;
```

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

Koszt pamięciowy możemy polepszyć do $M(n) = \Theta(1)$ stosując rekurencję ogonową:

```
let potega b n =
  let rec iter n a =
    if n = 0 then a else iter (n-1) (a*b)
  in
    iter n 1;;
```

przy czym `iter` $n\ a = a \cdot b^n$. Koszt czasowy pozostaje jednak bez zmian.

Możemy jednak skorzystać z innego wzoru na potęgowanie:

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

który zapisujemy jako:

```
let potega b n =
  let rec pot b n a =
    if n = 0 then a
    else if parzyste n then pot (square b) (n / 2) a
    else pot b (n - 1) (a * b)
  in
    pot b n 1;;
```

przy czym `pot` $b\ n\ a = a \cdot b^n$. Mamy tutaj rekurencję ogonową, więc koszt pamięciowy jest stały, $M(n) = \Theta(1)$. Jaka jest jednak liczba kroków? Można pokazać przez indukcję, że jeżeli 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 $T(n) = \Theta(\log n)$.

8.6 Przykład: stopień parzystości

[[Zmienić na logarytm całkowitoliczbowy.]]

Przypomnijmy sobie zadanie 1 z wykładu 2. Stopień parzystości liczby całkowitej x, to największa taka liczba naturalna i, że x dzieli się przez 2^i. Liczby nieparzyste mają stopień parzystości 0, liczby 2 i -6 mają stopień parzystości 1, a liczby 4 i 12 mają stopień parzystości 2. Przyjmujemy, że 0 ma stopień parzystości -1.

Standardowe rozwiązanie tego zadania ma następującą postać:

```
let rec stopien_parzystosci n =
  if n = 0 then -1
  else if n mod 2 = 0 then 1 + stopien_parzystosci (n/2)
  else 0;;
```

Z każdym krokiem iteracji n jest dzielone przez 2. Tak więc, dla $n \neq 0$, kroków tych nie będzie więcej niż $\log_2 |n| + 1$, przy czym ograniczenie to jest osiągane dla $n = 2^k$. Czyli złożoność czasowa jest rzędu $T(n) = O(\log |n|)$. Ponieważ rekurencja nie jest ogonowa, złożoność pamięciowa jest taka sama, jak czasowa $M(n) = \Theta(\log |n|)$.

Złożoność pamięciową łatwo poprawić do $M(n) = \Theta(1)$, stosując rekurencję ogonową:

```
let stopien_parzystosci n =
  let rec pom a n =
    if n mod 2 = 0 then pom (a+1) (n/2)
    else a
  in
    if n = 0 then -1 else pom 0 n;;
```

Nie jest to jednak rozwiązanie optymalne. Złożoność czasową można istotnie polepszyć. Pomyślmy o zapisie binarnym liczby równej stopniowi parzystości n. Powiedzmy, że najstarszy bit tej liczby najduje się na pozycji i. Liczbę i możemy wyznaczyć badając podzielność n przez liczby postaci $2^1, 2^2, 2^4, \ldots, 2^{2^i}, 2^{2^{i+1}}$. Kolejne bity wyznaczamy badając stopień parzystości liczby $\frac{n}{2^{2^i}}$.

```
let stopien_parzystosci n =
  let rec pom a k p n =
    if n mod 2 <> 0 then a
    else
      let s = p*p
      in
        if n mod s = 0 then pom a (2*k) s n
        else  (assert (n mod p = 0); pom (a+k) 1 2 (n / p))
  in
    if n = 0 then -1 else pom 0 1 2 n;;
```

Dla procedury pomocniczej `pom` spełniony jest następujący niezmiennik: $p = 2^k$ i $p \mid n$ lub $2 \nmid n$, oraz następujący warunek końcowy: `pom` $a\ k\ n = a +$ `stopien_parzystosci` n.

Wyznaczenie pozycji i najstarszego bitu wyniku wymaga $i + 1$ kroków. Wyznaczając kolejny bit wyniku nie korzystamy z wcześniej obliczanych wartości, tylko wyznaczamy go od początku, jako najstarszy bit liczby $\frac{|n|}{2^{2^i}}$, itd. Stąd złożoność czasowa algorytmu jest rzędu $T(n) = O((\log \log |n|)^2)$. Dzięki zastosowaniu rekurencji ogonowej złożoność pamięciowa jest rzędu $M(n) = \Theta(1)$.

Złożoność czasową można jeszcze polepszyć. Wyznaczając pozycję i najstarszego bitu wyniku obliczamy liczby $2^1, 2^2, 2^4, \ldots, 2^{2^i}, 2^{2^{i+1}}$. Jeśli je zapamiętamy, to możemy je wykorzystać do wyznaczenia kolejnych bitów wyniku.

```
let stopien_parzystosci n =
  let rec pom a k ((h::t) as l) n =
    if n mod 2 <> 0 then a
```

```
    else
      let s = h * h
      in
        if n mod s = 0 then pom a (2*k) (s::l) n
        else if n mod h = 0 then pom (a+k) (k/2) t (n/h)
        else pom a (k/2) t n
  in
    if n = 0 then -1 else pom 0 1 [2; 1] n;;
```

Dla procedury pomocniczej `pom` spełniony jest następujący niezmiennik: $l = [2^k; 2^{k-1}; ...; 2; 1]$, oraz następujący warunek końcowy: `pom` $a\ k\ l\ n = a +$ `stopien_parzystosci` n. Najpierw konstruowana jest lista l, aż do momentu gdy $k = i$, a następnie wartości z tej listy są używane do wyznaczenia kolejnych bitów wyniku. Złożoność czasowa jest rzędu $T(n) = \Theta(\log\log|n|)$. Ze względu na rozmiar listy l, złożoność pamięciowa jest również rzędu $M(n) = \Theta(\log\log|n|)$.

[[Dorobić rozwiązanie działające w czasie $\Theta\left(\frac{\log\log x}{\log y}\right)$ – oparte o liczby y^{Fib_i}.]]

8.7 Przykład: algorytm Euklidesa przez odejmowanie

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

Dla $x, y > 0$ mamy (`nwd` $x\ y$) $= NWD(x, y)$.

Jaka jest złożoność czasowa (ze względu na $n = x + y$)? Liczba wykonywanych kroków może być liniowa, np. dla (`nwd` x 1), czyli $T(n) = \Omega(n)$. Gorsza nie będzie, bo z każdym krokiem maleje wartość $x + y$, $T(n) = O(n)$, czyli $T(n) = \Theta(n)$. Mamy tu do czynienia z rekurencją ogonową, więc złożoność pamięciowa jest stała, $M(n) = \Theta(1)$.

8.8 Przykład: algorytm Euklidesa przez dzielenie

```
let nwd a b =
  let rec e a b =
    if b = 0 then a else e b (a mod b)
  in
    if a > b then e a b else e b a;;
```

Jakie są wymagania wobec argumentów? ($\max(a, b) > 0$, $\min(a, b) \geq 0$) Ile operacji arytmetycznych wykona ten algorytm (ze względu na $\min(a, b)$)?

Lemat 1 (Lame). *Oznaczmy przez (a_i, b_i) pary wartości a i b, dla których powyższy algorytm wykonuje i kroków. Wówczas $b_i \geq Fib_{i-1}$.*

Dowód. Dowód indukcyjny.

1. jeśli jeden krok, to $b_1 = 0$,

2. jeśli dwa kroki, to $b \geq 1$,

3. jeśli więcej kroków, to $(a_{k+1}, b_{k+1}) \to (a_k, b_k) \to (a_{k-1}, b_{k-1})$, 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 $T(a+b) = O(\log(a+b))$. 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 $T(a+b) = \Theta(\log(a+b))$.

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

8.9 Przykład: algorytm Euklidesa przez parzystość

Zastosowanie zasady dziel i zwyciężaj.

```
let nwd x y =
  let rec pom x y a =
    if x = y then a * x
    else if parzyste x && parzyste y then pom (x / 2) (y / 2) (2 * a)
    else if parzyste x then pom (x / 2) y a
    else if parzyste y then pom x (y / 2) a
    else if x > y then pom (x - y) y a else pom x (y - x) a
  in
    pom x y 1;;
```

Algorytm ten wykorzystuje jedynie odejmowanie i mnożenie/dzielenie/modulo 2. Ze względu na zapis binarny liczb operacje te mają koszt O(długość zapisu liczby), nawet dla bardzo dużych liczb. Jakie są wymagania wobec argumentów? $(a, b > 0)$ Ile operacji arytmetycznych wykona ten algorytm?

Oznaczmy przez (a_i, b_i) pary wartości a i b, dla których powyższy algorytm wykonuje i kroków. Mamy $a_{i+1} \geq a_i$, $b_{i+1} \geq b_i$, $b_1 = a_1 > 0$. W pierwszych trzech przypadkach $a_{i+1}b_{i+1} \geq 2a_ib_i$. W czwartym przypadku $a_{i+1}b_{i+1} \geq 2a_{i-1}b_{i-1}$. Przez indukcję pokazujemy, że

$$a_ib_i \geq 2^{\lfloor \frac{i-1}{2} \rfloor}$$

Stąd algorytm wykona $O(\log(ab)) = O(\log \max(a,b))$ kroków.

Z drugiej strony, $a_{i+1} + b_{i+1} \leq 2(a_i + b_i)$. Czyli $a_i + b_i \leq 2^{i-1}(a_1 + b_1)$. Stąd algorytm wykona $\Omega(\log(a+b)) = \Omega(\log \max(a,b))$ kroków. Tak więc algorytm wykona $\Theta(\log \max(a,b))$ kroków.

Każdy krok niesie ze sobą stały koszt czasowy, stąd koszt czasowy wynosi $T(\max(a,b)) = \Theta(\log \max(a,b))$. Mamy tu do czynienia z rekurencją ogonową, stąd stała złożoność pamięciowa, $M(\max(a,b)) = \Theta(1)$.

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

8.10 Przykład: Liczby Fibonacciego

Przyjrzyjmy się różnym algorytmom liczenia liczb Fibonacciego. Najprostszy z nich, i zarazem najmniej efektywny, opiera się na rekurencyjnym wzorze definiującym liczby Fibonacciego.

$$Fib_0 = 0 \qquad Fib_1 = 1 \qquad Fib_{n+1} = Fib_n + Fib_{n-1}$$

```
let rec fib n =
  if n < 2 then n else fib (n - 1) + fib (n - 2);;
```

Jaka jest złożoność czasowa tego rozwiązania? Wyobraźmy sobie, że obliczając Fib_n rozwijamy podaną definicję rekurencyjną, aż do uzyskania sumy zer i jedynek. Uzyskamy sumę zawierającą Fib_n jedynek i nie więcej niż Fib_n zer. Wiemy (z EMD), że $Fib_n \approx \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}}$. Tak więc złożoność czasowa tego algorytmu to $T(n) = \Theta(Fib_n) = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$. Ponieważ rekurencja w rozwiązaniu nie jest ogonowa, więc złożoność pamięciowa jest takiego rzędu, jak głębokość rekurencji, czyli $M(n) = \Theta(n)$.

Bardziej efektywne jest

- z parami i rekurencją ogonową,

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

- w koszcie czasowym rzędu$\Theta(\log n)$ i stałym pamięciowym (nie korzystając ze wzoru na n-tą liczbę Fibonacciego).

```
let mnoz ((x11, x12), (x21, x22)) ((y11, y12), (y21, y22)) =
  ((x11 * y11 + x12 * y21, x11 * y12 + x12 * y22),
   (x21 * y11 + x22 * y21, x21 * y12 + x22 * y22));;


let square x = mnoz x x;;


let f = ((0, 1), (1, 1));;


let id = ((1, 0), (0, 1));;
```

```
let potega b n =
  let rec pot b n a =
    if n = 0 then a
    else if parzyste n then pot (square b) (n / 2) a
    else pot b (n - 1) (mnoz a b)
  in
    pot b n id;;

let fib n =
  let ((_, v), _) = potega f n
  in v;;
```

8.11 Test na liczby pierwsze

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

```
let min_dzielnik n =
  let rec dziel k =
    if square k > n then n
    else if (n mod k) = 0 then k
    else dziel (k + 1)
  in
    dziel 2;;
```

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

8.12 Test Fermata i Millera-Rabina

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

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

Fakt 1. *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.*

Dowód. Niech p będzie tym nietrywialnym pierwiastkiem. Wówczas $(p-1)\cdot(p+1) = p^2-1 = 0$. Równocześnie $p-1 \neq 0$ i $p+1 \neq 0$, a więc Z_n nie jest ciałem, czyli n nie jest liczbą pierwszą. □

Nasz algorytm polega na wylosowaniu liczby a i sprawdzeniu, czy spełnione jest tw. Fermata. Ponadto obliczając $a^{n-1} \bmod n$ sprawdzamy, czy „mod n" to ciało. Jeżeli nie, to n nie może być liczbą pierwszą.

Jeśli jest to ciało, to mamy dwa $\sqrt{1}$: 1 i $n-1$. Jeżeli znajdziemy inny (nietrywialny) $\sqrt{1}$, to nie jest to ciało i n nie może być liczbą pierwszą.

Przykład: Weźmy $n = 15$. Ewidentnie $15 = 3{\cdot}5$ nie jest liczbą pierwszą. Mamy następujące pierwiastki jedynki: $1^2 = 1 \equiv 1 \bmod 15$, $4^2 = 16 \equiv 1 \bmod 15$, $11^2 = 121 \equiv 1 \bmod 15$, $14^2 = 196 \equiv 1 \bmod 15$. Przy tym 4 i 11 nie są trywialnymi pierwiastkami z 1.

Obliczając $a^{n-1} \bmod n$ sprawdzamy wszystkie pojawiające się wartości, czy nie są nietrywialnymi pierwiastkami z 1. Przypadki, gdy natrafiamy na taki pierwiastek sygnalizujemy podnosząc wyjątek.

```
exception Pierwiastek;;

let rec expmod b k n =
  let test x =
    if not (x = 1) && not (x = n - 1) && (square x) mod n = 1 then
      raise Pierwiastek
    else
      x
  in
    if k = 0 then 1 else
    if parzyste k then (square (test (expmod b (k / 2) n))) mod n
    else ((test (expmod b (k-1) n)) * b) mod n;;
```

Można pokazać, że jeżeli n jest liczbą nieparzystą i nie jest liczbą pierwszą, to przynajmniej dla połowy $1 < a < n - 1$ obliczanie $a^{n-1} \bmod n$ odkryje nietrywialny pierwiastek 1. Tak poprawiony test Fermata jest znany jako test Millera-Rabina.

```
let randtest n =
  if parzyste n then
    n = 2
  else if n = 3 then true
  else
    try
      expmod (Random.int (n-3) + 2) (n-1) n = 1
    with Pierwiastek -> false;;
```

Z pewnym prawdopodobieństwem test ten może stwierdzić, że liczba, która nie jest pierwsza, jest pierwsza. Spróbujmy oszacować prawdopodobieństwo pomyłki. Załóżmy, że n nie jest liczbą pierwszą i $n > 3$. Dodatkowo załóżmy, że dla wylosowanego a spełnione jest kryterium Fermata. Prawdopodobieństwo pomyłki możemy oszacować z góry prawdopodobieństwem pomyłki testu na nietrywialne pierwiastki z 1. To prawdopodobieństwo nie przekracza natomiast $\frac{1}{2}$. Tak więc test ten z prawdopodobieństwem co najmniej $\frac{1}{2}$ daje poprawną odpowiedź (a tak naprawdę z dużo większym), a z prawdopodobieństwem nie przekraczającym $\frac{1}{2}$ uzna liczbę, która nie jest pierwsza, za pierwszą.

Powtarzając taki test k zmniejszamy prawdopodobieństwo błędu poniżej $(\frac{1}{2})^k$. Chcąc, aby prawdopodobieństwo testu spadło poniżej ε należy go powtórzyć $\lceil \log_{\frac{1}{2}} \varepsilon \rceil$.

Gadka o algorytmach randomizacyjnych. Dwie klasy:

- Monte Carlo — złożoność zawsze dobra, ale z małym prawdopodobieństwem może dawać złe wyniki, lub wyniki nieznacznie zaburzone,

- Las Vegas — zawsze daje dobre wyniki, ale z małym prawdopodobieństwem działa dłużej; średnia złożoność musi być OK.

Jeśli chcemy być pewni wyniku z prawdopodobieństwem p, to powtarzamy ten test $-\log_2(1-p)$ razy. Koszt pojedynczego testu, pamięciowy i czasowy, jest rzędu $\Theta(\log n)$. Ponieważ liczba powtórzeń testu jest stała, więc koszt całego algorytmu jest rzędu $\Theta(\log n)$.

Ćwiczenia

1. Algorytm mnożenia rosyjskich chłopów.

2. Dana jest para funkcji f i g, z liczb całkowitych w liczby całkowite. Wiadomo, że f jest ściśle rosnąca, a g jest ściśle malejąca. Napisz taką procedurę `znajdź : (int → int) → (int → int) → int`, że dla $w =$ `znajdź` $f\ g$, mamy:

$$|f(w) - g(w)| = \min_{i \in \mathbb{Z}} |f(i) - g(i)|$$

   Możesz założyć, że istnieją takie liczby całkowite i i i', że $f(i) < g(i)$ oraz $f(i') > g(i')$.

   Podaj złożoność czasową i pamięciową swojego rozwiązania.

3. [CEOI 2003, uproszczone zadanie Hanoi] Wiadomo (z EMD), że żeby przełożyć n krążków w łamigłówce „wieże Hanoi" trzeba wykonać $2^n - 1$ ruchów. Napisz procedurę `hanoi: int list -> int -> int`, która dla zadanej konfiguracji krążków oraz słupka, na którym początkowo znajdują się wszystkie krążki wyznaczy minimalną liczbę ruchów potrzebnych do uzyskania danej konfiguracji.

   Słupki są reprezentowane jako liczby całkowite od 1 do 3. Konfiguracja to lista numerów słupków, na których mają się znaleźć krążki, w kolejności od największych krążków do najmniejszych.

4. Chiński łańcuch (przynieść i pokazać). W jednym ruchu można zawsze zdjąć lub założyć pierwsze ogniwo łańcucha. Ponadto, jeżeli zdjęte są ogniwa o numerach $1, 2, \ldots, k-2$, ogniwo nr $k-1$ jest założone, to w jednym ruchu można zdjąć lub założyć ogniwo nr k. Początkowo wszystkie ogniwa łańcucha są założone. Napisz procedurę, której wynikiem jest lista ruchów prowadzących do zdjęcia n ogniw łańcucha. Oblicz złożoność czasową i pamięciową rozwiązania. Jeśli złożoność czasowa jest większa od pamięciowej ($\Theta(2^n)$), to popraw program używając akumulatora i nie używając sklejania list.

Wykład 9. 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 sortowania.

Eliminacja algorytmów randomizowanych Jeśli nasz algorytm 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 2. *Jeśli drzewo binarne ma k liści, to ma wysokość nie mniejszą niż $\log_2 k$. (Dowód na ćwiczenia.)*

Lemat 2. $3^n \cdot n! \geq n^n$

Dowód. Indukcyjnie. $3 \geq 1$.

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

□

Tw. 2. *Pesymistyczny koszt sortowania jest rzędu $\Omega(n \log n)$.*

Dowód. Rozpatrujemy wszystkie możliwe wykonania naszego algorytmu dla ciągu n różnych elementów. Przebieg obliczenia zależy tylko od porównań tych elementów. Wszystkie takie obliczenia możemy przedstawić sobie w postaci drzewa decyzyjnego, w którego węzłach mamy porównania danych elementów.

Drzewo to ma co najmniej $n!$ liści, gdyż mamy $n!$ możliwych wyników. Z lematu i faktu wynika, że wysokość drzewa h

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

□

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

9.1 Selection sort

Idea algorytmu — redukcja = wybór maksimum. Realizacja:

```
let select_max (h::t) =
  fold_left
    (fun (m, r) x -> if x > m then (x, m::r) else (m, x::r))
    (h, []) t;;
```

```
let selection_sort l =
  let przenies (s, l) =
    let (m, r) = select_max l
    in (m::s, r)
  in
    iteruj
      ([], l)
      przenies
      (fun (_, l) -> l = [])
      (fun (s, _) -> s);;
```

Analiza kosztu.

9.2 Insertion sort

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

```
let wstaw l x =
  (filter (fun y -> y <= x) l) @
  (x :: (filter (fun y -> y > x) l));;

let insertion_sort l =  fold_left wstaw [] l;;
```

Złożoność czasowa `wstaw` jest liniowa ze względu na długość przetwarzanej listy, a złożoność czasowa `insertion_sort` jest rzędu $\Theta(n^2)$. Złożoność pamięciowa jest rzędu $\Theta(n)$, gdyż w każdej chwili pamiętamy tylko kilka list o długościach nie przekraczających n.

Można też poprawić trochę złożoność tego algorytmu. Oznaczmy przez i liczbę inwersji w danym ciągu. Przyjrzyjmy się następującej wersji sortowania przez wstawianie:.

```
let wstaw x l =
  let rec wst a x l =
    match l with
      []   -> rev(x::a) |
      h::t ->
        if x > h then wst (h::a) x t
        else rev (x::a) @ l
  in wst [] x l;;

let insertion_sort l =  fold_right wstaw l [];;
```

W każdym kroku iteracji w procedurze `wst` pozbywamy się jednej inwersji związanej z elementem x. Stosując taki algorytm wstawiania otrzymujemy sortowanie działające w czasie $\Theta(n+i)$. Oczywiście liczba inwersji może być rzędu $\Theta(n^2)$. W następnym punkcie zobaczymy zastosowanie takiej implementacji sortowania przez wstawianie.

9.3 Sortowanie Shell'a

She shall sort sea shells with a Shell-sort.
...Sortowanie Shell'a to uogólnienie sortowania przez wstawianie. Składa się on z wielu faz. W każdej fazie dzielimy ciąg na k krótszych ciągów, każdy złożony z co k-tych elementów. Na przykład dla $k = 2$ mamy dwa ciągi, jeden złożony z elementów na parzystych pozycjach i jeden złożony z elementów na nieparzystych elementów. Każdy z k ciągów sortujemy niezależnie, za pomocą sortowania przez wstawianie.

Na fazę sortowania można też spojrzeć tak: Elementy sortowanego ciągu wpisujemy wierszami do tabeli o k kolumnach, następnie każdą kolumnę sortujemy za pomocą sortowania przez wstawianie, po czym odczytujemy elementy wierszami.

W rezultacie uzyskujemy ciąg $(x_1, \ldots, x_n)$, który jest k-posortowany, tzn. $x_i \leq x_{i+k}$ dla $1 \leq i \leq n - k$. Naszym celem jest uzyskanie ciągu posortowanego, czyli 1-posortowanego. W kolejnych fazach parametr k jest coraz mniejszy, aż w ostatniej fazie mamy $k = 1$. Tym samym oczywiste jest, że uzyskujemy ciąg posortowany. Po co więc poprzedzające fazy? Koszt sortowania przez wstawianie jest niewielki, jeżeli sortujemy ciąg, który jest już „prawie posortowany". Idea sortowania Shella polega na tym, że w każdej kolejnej fazie sortujemy przez wstawianie właśnie takie ciągi, które są „prawie posortowane".

9.4 Sortowanie przez scalanie

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

```
let rec merge_sort l =
  let split l =
    fold_left (fun (l1, l2) x -> (l2, x::l1)) ([], []) l
  and merge l1 l2 =
    let rec mrg a l1 l2 =
      match l1 with
        [] -> (rev a) @ l2 |
        (h1::t1) ->
          match l2 with
            [] -> (rev a) @ l1 |
            (h2::t2) ->
              if h1 > h2 then
                mrg (h2::a) l1 t2
              else
                mrg (h1::a) t1 l2
    in
      mrg [] l1 l2
  in
    match l with
      [] -> []|
      [x] -> [x]|
      _ ->
        let
          (l1, l2) = split l
        in
```

```
merge (merge_sort l1) (merge_sort l2);;
```

Analiza złożoności.

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.

$$T(1) = 1$$

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

Czyli dla $n = 2^k$ mamy:

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

Można pokazać, że $T(n)$ jest monotoniczna, a więc $T(n) = \Theta(n \log n)$ dla dowolnych n.

Ćwiczenia

1. Maksymalne plateau w liście (bez rekurencji).

2. Tomek ma zabawkę, z której wystają drewniane słupki różnej wysokości. Jednym uderzeniem młotka może wbić lub wysunąć wybrany słupek o 1.

   Napisz procedurę `słupki`, która dla danej listy początkowych wysokości słupków obliczy minimalną liczbę uderzeń młotka potrzebnych do wyrównania wysokości słupków.

3. Dysponujemy pewną pulą wolnych bloków pamięci. W tych blokach musimy umieścić pewien zestaw zmiennych, przy czym:

   - w jednym bloku może znajdować się co najwyżej jedna zmienna,
   - blok, w którym znajduje się dana zmienna nie może być od niej mniejszy.

   Napisz procedurę `da_się: int list → int list → bool`, która na podstawie listy wielkości wolnych bloków oraz listy wielkości zmiennych ustali, czy da się rozmieścić zmienne w blokach.

4. Napisz procedurę `przedział : int list -> int -> int`, która dla danej listy $[x_1; \ldots; x_n]$ oraz dla liczby całkowitej $r \geq 0$ obliczy taką liczbę całkowitą c, że $|\{i : |x_i - c| \leq r\}|$ jest maksymalne.

   Przykład: `przedział [2; -2; 5; -1; 11; 8; 4; 5; 8; 7] 2 = 6`.

5. Napisz funkcję `elementy` : α `list` → `int list` → α `list`, która dla list $[x_1; x_2; \ldots, x_n]$ i $[y_1; y_2; \ldots; y_m]$ zwraca listę $[x_{y_1}; x_{y_2}; \ldots; x_{y_m}]$. Możesz założyć, że liczby całkowite y_i w drugiej liście są z przedziału $[1, n]$, gdzie n oznacza długość pierwszej listy.

6. Zadanie o katastrofach lotniczych: Dana jest lista liczb całkowitych $[x_1; \ldots; x_n]$. Dla każdego i trzeba policzyć największe takie k_i, że $x_i = \max(x_{i-k_i+1}, \ldots x_i)$. Przyjmujemy, że $x_0 = \infty$.

7. Dana jest lista $[x_1; \ldots; x_n]$. Dla $1 \leq i < j \leq n$ powiemy, że elementy x_i i x_j widzą się nawzajem, jeżeli $\min(x_i, x_j) \geq \max(x_{i+1}, \ldots, x_{j-1})$.

   Napisz procedurę `widoczne`, która obliczy ile elementów ciągu jest widocznych z kolejnych jego elementów. Np. `widoczne [1; 8; 5; 6; 4] = [1; 3; 2; 3; 1]`.

8. Dana jest lista liczb całkowitych $[x_1; \ldots; x_n]$. Dla każdego i trzeba policzyć największe takie otoczenie x_i, czyli podciąg $x_{i-k_i}, \ldots x_{i+k_i}$, że $x_i = \max(x_{i-k_i}, \ldots x_{i+k_i})$.

9. [II OI] Napisz procedurę `trójkąt : int list → bool`, która dla danej listy $[x_1; x_2; \ldots; x_n]$ dodatnich liczb całkowitych sprawdzi, czy lista zawiera trójkę elementów x_i, x_j i x_k (dla $i \neq j$, $j \neq k$, $i \neq k$) spełniających nierówność trójkąta, tzn.:

   $$2 \cdot \max(x_i, x_j, x_k) \leq x_i + x_j + x_k$$

   Podaj złożoność czasową i pamięciową swojego rozwiązania. (Uwaga: Jeżeli liczby całkowite mają ograniczony zakres, to można to zadanie rozwiązać w stałym czasie.)

10. Dana jest (niepusta) lista $[x_1; x_2; \ldots; x_n]$ oraz liczba d. Szukamy takich elementów x_i i x_j, dla których wartość $|x_i - x_j|$ jest jak najbliższa d (tzn. wartość $||x_i - x_j| - d|$ jest minimalna).

    Napisz procedurę `różnica : int list → int → int`, która dla danej listy oraz wartości d wyznacza różnicę elementów $|x_i - x_j|$ najbliższą d.

    Podaj złożoność czasową i pamięciową swojego rozwiązania.

11. [XII OI, zadanie Sumy Fibonacciego] System Zeckendorfa, to taki system, w którym liczba jest reprezentowana jako ciąg zer i jedynek, a kolejne cyfry mają takie znaczenie, jak kolejne liczby Fibonacciego $(1, 2, 3, 5, \ldots)$. Ponadto, w reprezentacji liczby nie mogą pojawiać się dwie jedynki obok siebie.

    Liczby reprezentujemy jako listy zer i jedynek. Zaimplementuj dodawanie w systemie Zeckendorfa.

Wytyczne dla prowadzących ćwiczenia

Ad. 2 Do wyznaczenia mediany należy użyć sortowania.

Ad. 6 Wymaga użycia stosu.

Ad. 7 Rozszerz rozwiązanie zadania 6.

Ad. 8 Rozszerz rozwiązanie zadania 6.

Ad. 11 Stosunkowo łatwo można rozwiązać to zadanie w czasie $O(n^2)$, a dużo trudniej w czasie $O(n)$.

Wykład 10. Zasada dziel i zwyciężaj na przykładzie sortowania cd.

10.1 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.

```
let rec quick_sort l =
  let split l x = (filter (fun y -> y < x) l,
                   filter (fun y -> y = x) l,
                   filter (fun y -> y > x) l)
  in
    if length l < 2 then l else
      let  x = nth l (Random.int (length l))
      in
        let (ll, le, lg) = split l x
      in
      (quick_sort ll) @ le @ (quick_sort lg);;
```

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

$$M(0) = 1$$

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

$$T(0) = 1$$

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

Zbadajmy jednak jak sprawa wygląda średnio.

Tw. 3. *Algorytm Quick-sort, dla 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 3. *Niech wybrany element sortowanej listy jest równy* s. *Listy, na które jest rozdzielana dana lista są losowymi permutacjami zbiorów* $\{1, \ldots, s-1\}$, $\{s\}$ *i* $\{s+1, \ldots, n\}$.

Dowód lematu. Dla ustalonego s, na wejściu możemy mieć $n!$ możliwych list. Element s może być na n pozycjach, na każdej z równym prawdopodobieństwem, czyli dla ustalonej pozycji i wartości elementu s pozostaje nam $(n-1)!$ możliwych permutacji. Wystarczy więc pokazać, że każda para wynikowych permutacji odpowiada dokładnie:

$$\frac{(n-1)!}{(s-1)! \cdot 1 \cdot (n-s)!} = \binom{n-1}{s-1}$$

różnych permutacji na wejściu. Zauważmy, że dla ustalonych list wynikowych, podział $n-1$ elementów (poza s) w liście wejściowej na te, które trafiły do pierwszej i ostatniej listy jednoznacznie wyznaczają listę wejściową. Takich rozdziałów jest jednak dokładnie

$$\binom{n-1}{s-1}$$

□

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. Zauważmy, że $C_0(z) = C_1(z) = 1$.

Koszt fazy algorytmu wynosi $3 \cdot n$, a wielkość wywoływanych faz wynosi odpowiednio $s-1$ i $n-s$, czyli:

$$\sum_{k\geq 0} c_{n,k,s} z^k = \sum_{k\geq 3n} z^k \cdot \left(\sum_{i=0}^{k-3n} c_{s-1,i} \cdot c_{n-s,k-3n-i} \right) = z^{3n} \cdot \sum_{k\geq 0} z^k \cdot \left(\sum_{i=0}^{k} c_{s-1,i} \cdot c_{n-s,k-i} \right) =$$

$$= z^{3n} \cdot \sum_{k\geq 0} \sum_{i=0}^{k} \left(z^i c_{s-1,i} \cdot z^{k-i} c_{n-s,k-i} \right) = z^{3n} \cdot \sum_{j\geq 0} \sum_{i\geq 0} \left(z^i c_{s-1,i} \cdot z^j c_{n-s,j} \right) =$$

$$= z^{3n} \cdot \left(\sum_{i\geq 0} c_{s-1,i} z^i \right) \cdot \left(\sum_{j\geq 0} c_{n-s,j} z^j \right) = z^{3n} C_{s-1}(z) C_{n-s}(z)$$

Ponieważ różne s są równoprawdopodobne, więc:

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

Czyli

$$C_n(z) = \sum_{k\geq 0} c_{n,k} z^k = \sum_{k\geq 0} \sum_{s=1}^{n} \frac{c_{n,k,s} z^k}{n} = \sum_{s=1}^{n} \frac{\sum_{k\geq 0} c_{n,k,s} z^k}{n} = \sum_{s=1}^{n} \frac{z^{3n} C_{s-1}(z) C_{n-s}(z)}{n}$$

Stąd mamy zależność rekurencyjną:

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

$$C_n(z) = \frac{1}{n} z^{3n} \sum_{s=1}^{n} C_{s-1}(z) C_{n-s}(z)$$

Nas interesuje tylko $C'_n(1)$, którą będziemy oznaczać w skrócie przez C'_n. Zauważmy, że $C'_0 = C'_1 = 0$. Ponieważ $C_n(1) = 1$, mamy:

$$C'_n = \frac{1}{n}(3n)1^{3n-1}\sum_{s=1}^{n} C_{s-1}(1)C_{n-s}(1) + \frac{1}{n}1^{3n}\sum_{s=1}^{n}\left(C'_{s-1}C_{n-s}(1) + C_{s-1}(1)C'_{n-s}\right) =$$

$$= \frac{1}{n}(3n)n + \frac{1}{n}\sum_{s=1}^{n}\left(C'_{s-1} + C'_{n-s}\right)$$

Otrzymujemy więc tożsamość:

$$C'_n = 3n + \frac{1}{n}\sum_{s=1}^{n}\left(C'_{s-1} + C'_{n-s}\right) = 3n + \frac{1}{n}\sum_{s=1}^{n} 2C'_{s-1} = 3n + \frac{2}{n}\sum_{s=0}^{n-1} C'_s$$

która intuicyjnie oddaje fakt, że w ramach każdego wywołania rekurencyjnego wykonujemy $3n$ porównań, plus porównania w dwóch wywołaniach rekurencyjnych. Przy tym wszystkie możliwe podziały $n-1$ elementów na dwa krótsze ciągi do posortowania są równoprawdopodobne.

Ponieważ $C'_0 = 0$, mamy stąd:

$$nC'_n = 3n^2 + 2\sum_{s=1}^{n-1} C'_s$$

stąd zaś:

$$nC'_n - (n-1)C'_{n-1} = 3n^2 - 3(n-1)^2 + 2\sum_{s=1}^{n-1} C'_s - 2\sum_{s=1}^{n-2} C'_s = 6n - 3 + 2C'_{n-1}$$

tak więc:

$$nC'_n = (n+1)C'_{n-1} + 6n - 3$$

czyli:

$$\frac{C'_n}{n+1} = \frac{C'_{n-1}}{n} + \frac{6n-3}{n(n+1)} = \frac{C'_{n-1}}{n} + \frac{6}{(n+1)} - \frac{3}{n(n+1)}$$

Ponieważ $C'_0 = C'_1 = 0$, więc:

$$\frac{C'_n}{n+1} \leq \frac{6}{n+1} + \frac{6}{n} + \cdots + \frac{6}{3} = 6(H_{n+1} - \frac{3}{2})$$

Czyli:

$$C'_n \leq 6(n+1)(H_{n+1} - \frac{3}{2})$$

Ponieważ:

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

gdzie γ jest stałą Eulera, więc:

$$E(C_n) = C'_n \leq 6(n+1)(\ln(n+1) + O(1)) = 6n\ln(n+1) + O(n) = \Theta(n\log n)$$

□

10.2 Heap-sort

10.2.1 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 multi-zbiory) wartości elementów. Dostępne są następujące operacje i pojęcia dotyczące takiej kolejki:

- `type 'a pri_queue` — typ kolejek,
- `empty_queue : 'a pri_queue` — pusta kolejka,
- `empty : : 'a pri_queue -> bool` — predykat sprawdzający czy kolejka jest pusta,
- `put : 'a pri_queue -> 'a -> 'a pri_queue` — włóż element do kolejki,
- `getmax : 'a pri_queue -> ('a * 'a pri_queue)` — wyjmij z kolejki największy element,
- `exception Empty_Queue` — wyjątek podnoszony, gdy próbujemy wyjąć element z pustej kolejki.

10.2.2 Sortowanie

Mając dostępną taką kolejkę możemy zrealizować np. sortowanie:

```
let heap_sort l =
  let wloz = fold_left put empty_queue l
  and wyjmij (l, q) =
    let (m, r) = getmax q
    in (m::l, r)
  in
    iteruj
      ([], wloz)
      wyjmij
      (fun (_, q) -> empty q)
      (fun (l, _) -> l)
```

10.2.3 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.

```
type 'a pri_queue = 'a list
let empty_queue = []
let empty q = q = []
let put q x = x::q
let getmax q =
```

```
  match q with
    [] -> raise Empty_Queue |
    _ -> select_max q
```

(Procedura `select_max` pochodzi z sortowania przez wybieranie.) Algorytm sortowania oparty na takiej realizacji kolejki, to sortowanie przez wybieranie.

10.2.4 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 funkcja 1–1.

```
type 'a pri_queue = 'a list
let empty_queue = []
let empty q = q = []
let put q x =
  (filter (fun y -> y > x) q) @
  (x :: (filter (fun y -> y <= x) q))
let getmax q =
  match q with
    [] -> raise Empty_Queue |
    h::t -> (h, t)
```

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

10.2.5 Stóg

Kolejkę priorytetową można pamiętać efektywniej, w postaci tzw. stogu. Stóg jest to drzewo binarne, w którego węzłach są przechowywane liczby i w którym dla każdego węzła spełniony jest następujący warunek: wartość przechowywana w węźle jest większa lub równa wszystkim wartościom przechowywanym w poddrzewie, którego ten węzeł jest korzeniem. Stóg reprezentujemy w następujący sposób:

```
type 'a pri_queue =
  Leaf |
  Node of 'a * 'a pri_queue * 'a pri_queue * int
```

Operacje na stogu implementujemy następująco:

```
let empty_queue = Leaf

let empty q = q = Leaf

let size q =
  match q with
    Leaf -> 0 |
    Node (_, _, _, n) -> n
```

```
let rec put h x =
  (* Pomocniczy selektor. *)
  let get_root (Node (r, _, _, _)) = r
  (* Pomocniczy modyfikator. *)
  and set_root (Node (_, l, p, n)) r = Node (r, l, p, n)
  in
    let rotate h =
      match h with
        Leaf -> h |
        Node (r, Leaf, Leaf, n) -> h |
        Node (r, l, Leaf, n) ->
          if r >= get_root l then h
          else Node (get_root l, set_root l r, Leaf, n)|
        Node (r, Leaf, p, n) ->
          if r >= get_root p then h
          else Node (get_root p, Leaf, set_root p r, n) |
        Node (r, l, p, n) ->
          let rl = get_root l
          and rp = get_root p
          in
            let (r1, rl1, rp1) =
              if r >= rl then
                if r >= rp then (r, rl, rp) else (rp, rl, r)
              else
                if rl >= rp then (rl, r, rp) else (rp, rl, r)
            in
              Node (r1, set_root l rl1, set_root p rp1, n)
      in
  match h with
    Leaf -> Node (x, Leaf, Leaf, 1) |
    Node (r, l, p, n) ->
      if size l <= size p then
        rotate (Node (r, put l x, p, n + 1))
      else
        rotate (Node (r, l, put p x, n + 1))

let getmax h =
  let rec del h =
    match h with
      Leaf -> h |
      Node (r, Leaf, Leaf, n) -> Leaf |
      Node (r, l, Leaf, n) -> l |
      Node (r, Leaf, p, n) -> p |
      Node (r, (Node (rl, _, _, _) as l),
                (Node (rp, _, _, _) as p), n) ->
        if rl >= rp then
          Node (rl, del l, p, n - 1)
```

```
      else
        Node (rp, l, del p, n - 1)
in
  match h with
    Leaf -> raise Empty_Queue|
    Node (x, _, _, _)  ->
    (x, del h)
```

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

Fakt 3. *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 $\Theta(n)$.

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

Bardziej formalnie, postępujemy następująco:

- określamy *funkcję 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 wartością funkcji amortyzacji dla końcowej i początkowej wartości struktury danych.

10.4 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.

```
type 'a fifo = {front: 'a list; back: 'a list; size: int};;

let empty_queue = {front=[]; back=[]; size=0};;

let size q = q.size;;

let is_empty_queue q = size q = 0;;

let balance q =
  match q with
    {front=[]; back=[]}        -> q |
    {front=[]; back=b; size=s} -> {front=rev b; back=[]; size=s} |
    _                          -> q;;

let put {front=f; back=b; size=n} x =
  balance {front=f; back=x::b; size=n+1};;
```

```
let first q =
  match q with
    {front=[]}   -> failwith "Pusta kolejka" |
    {front=x::_} -> x;;

let remove q =
  match q with
    {front=_::f} -> balance {front=f; back=q.back; size=q.size-1} |
    _            -> empty_queue;;
```

Zanalizujmy koszt zamortyzowany operacji. Funkcja amortyzacji to `length q.back`. Procedury `is_empty_queue`, `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 `remove` może mieć koszt liniowy, ze względu na procedurę `balance`. Jednak ta ostatnia wykonuje tyle kroków, o ile zmniejsza funkcję amortyzacji. Tak więc koszt zamortyzowany procedury `remove` jest stały.

Ćwiczenia

1. Sformułuj warunek określający, czy element ma być wstawiony do lewego, czy prawego poddrzewa, aby kolejne elementy były wstawiane na kolejnych poziomach od lewej do prawej.

2. Napisz funkcję, która przekształci zadane drzewo w stóg, zachowując jego kształt (w rozwiązaniu można wykorzystać zmodyfikowaną procedurę `rotate` z wykładu). Jaka jest złożoność tej procedury? W jaki sposób zależy ona od kształtu drzewa?

3. Napisz funkcję, która „wyważy" zadany stóg, to znaczy tak poprzestawia w nim elementy, aby miał minimalną wysokość. Jaka jest jej złożoność?

4. Rozszerzyć implementację kolejek FIFO o wkładanie i wyjmowanie elementów z obydwu stron (w koszcie zamortyzowanym stałym).

5. Zaimplementuj k-max-kolejkę, czyli kolejkę, do której można wkładać elementy i dowiadywać się jakie jest maksimum z k ostatnio włożonych elementów. Dokładniej, powinny być dostępne następujące operacje na k-max-kolejkach:

   - `init` : `int` $\rightarrow \alpha$ `max_queue` — tworzy nową pustą kolejkę z określonym parametrem k,
   - `put` : α `max_queue` $\rightarrow \alpha \rightarrow \alpha$ `max_queue` — wkłada element do kolejki,
   - `get_max` : α `max_queue` $\rightarrow \alpha$ — zwraca minimum z k ostatnich elementów włożonych do kolejki.

   Wszystkie operacje na kolejce powinny działać w stałym czasie zamortyzowanym.

6. Mamy daną listę liczb całkowitych $l = [a_1, \ldots, a_n]$. Liczbę a_i nazywamy k-maksimum, jeżeli jest ona większa od $a_{max(i-k,1)}, \ldots, a_{i-1}, a_{i+1}, a_{min(i+k,n)}$. Napisz funkcję `kmax: int` $\rightarrow$ `int list` $\rightarrow$ `int list`, dla której wynikiem `kmax` k l jest podlista listy l złożona z samych k-maksimów.

7. [SICP, Ćw 2.29] Mobil, to ruchoma konstrukcja o budowie rekurencyjnej. Mobil ma ruchome ramiona, jak w wadze szalkowej, określonej długości. Na końcu każdego z ramion zawieszony jest ciężarek o określonej masie, lub mniejszy mobil.

   - Zaimplementuj strukturę danych reprezentującą mobile.
   - Napisz procedurę obliczającą wagę mobila.
   - Napisz procedurę sprawdzającą, czy mobil jest wyważony.

   * [BOI 1999 — uproszczone] Napisz procedurę, która zrównoważy dany mobil tak, aby jego odważniki miały dodatnie wagi całkowite, a jego całkowita masa była minimalna.

Wytyczne dla prowadzących ćwiczenia

Ad. 5 Rozwiązując to zadanie należy skorzystać z rozwiązania zadania 4 i zadania 6 z poprzedniego wykładu.

Ad. 6 Należy skorzystać z rozwiązania zadania 5.

Wykład 11. Moduły

11.1 Wprowadzenie

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

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

Moduł ma charakter czarnej skrzynki (ang. *black-box approach*). Na zewnątrz modułu widoczne są wyłącznie te obiekty programistyczne, które tworzą interfejs. Natomiast sposób ich implementacji, jak i ew. obiekty pomocnicze są ukryte wewnątrz modułu. Specyfikacja modułu nie powinna odwoływać się do sposobu implementacji modułu, a jedynie do takich właściwości modułu, które może zaobserwować użytkownik modułu. Co więcej, użytkownik nie tylko nie ma możliwości, ale i nie powinien (w swym programie) wnikać w sposób implementacji modułu (ang. *information hiding*).

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

W Ocamlu możemy osobno definiować interfejsy i implementacje modułów. Interfejsy modułów nazywamy *sygnaturami* natomiast ich implementacje *strukturami*. Dodatkowo istnieje pojęcie modułów sparametryzowanych — *funktorów*. Są to moduły, które mają określony interfejs, a także korzystają z modułów o określonych interfejsach, ale nie sprecyzowanej implementacji. Dopiero po podaniu implementacji uzyskujemy wynikowy moduł. Funktory można traktować jak konstrukcje przekształcające moduły w moduły.

11.2 Proste struktury

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

⟨definicja⟩ ::= `module` ⟨Identyfikator⟩ `=` ⟨struktura⟩
⟨struktura⟩ ::= `struct` { ⟨definicja⟩ }* `end`

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

⟨identyfikator⟩ ::= ⟨Identyfikator⟩ `.` ⟨identyfikator⟩

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

⟨jednostka kompilacji⟩ ::= open ⟨Identyfikator⟩

Przykład: Proste przykłady modułów:

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

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

module Pusty =
  struct
  end;;
```

Przykład: Moduły mogą zwierać wewnątrz moduły:

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

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

11.3 Sygnatury

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

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

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

Przykład: Taka sobie sygnatura:

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

Przykład: Sygnatura kolejek FIFO:

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

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

Przykład: Sygnatura kolejek dwustronnych:

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

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

Przykład:

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

11.4 Łączenie struktur i sygnatur

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

⟨definicja⟩ ::= module ⟨Identyfikator⟩ [: ⟨sygnatura⟩] = ⟨struktura⟩
⟨struktura⟩ ::= ⟨struktura⟩ : ⟨sygnatura⟩
⟨struktura⟩ ::= ⟨Identyfikator⟩
⟨sygnatura⟩ ::= ⟨Identyfikator⟩

Przykład: Różne sposoby definiowania modułu FIFO:

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

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

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

Podobnie jak można rozszerzać sygnatury, można też rozszerzać moduły:

Przykład: Liczby wymierne z dwiema barierami abstrakcji. Zwróćmy uwagę, że operacje arytmetyczne są zaimplementowane bez znajomości implementacji struktury danych ułamków.

```
module type ULAMKI =
  sig
    type t
    val ulamek : int -> int -> t
    val licznik : t -> int
    val mianownik : t -> int
  end;;

module type RAT =
  sig
    include ULAMKI
    val plus : t -> t -> t
    val minus : t -> t -> t
    val razy : t -> t -> t
    val podziel : t -> t -> t
    val rowne : t -> t -> bool
  end;;

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

module Rat : RAT =
    struct
      include Ulamki
      let plus x y =
        ulamek
          (licznik x * mianownik y + licznik y * mianownik x)
          (mianownik x * mianownik y)
      let minus x y =
        ulamek
```

```
      (licznik x * mianownik y - licznik y * mianownik x)
      (mianownik x * mianownik y)
  let razy x y =
    ulamek
      (licznik x * licznik y)
      (mianownik x * mianownik y)
  let podziel x y =
    ulamek
      (licznik x * mianownik y)
      (mianownik x * licznik y)
  let rowne x y =
    (licznik x * mianownik y) = (licznik y * mianownik x)
end;;
```

11.5 Jak wyodrębniać moduły?

Jakimi zasadami kierować się dzieląc program na moduły? Każdy program można podzielić na moduły: pierwsze 100 linii, drugie 100 linii, itd. Oczywiście nie każdy podział jest właściwy. Podział programu na moduły powinien być taki, aby:

- powiązania między modułami były jak najmniejsze; jak najmniej szczegółów budowy jednego modułu miało wpływ na budowę innego modułu,
- jeden moduł powinien koncentrować się na jednej decyzji projektowej, jednym „sekrecie"; nie należy łączyć nie związanych ze sobą sekretów w jednym module.

Powyższe zasady są znane pod nazwą *separation of concerns*.

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

Przykład: [D.L.Parnas, *On the Criteria To Be Used in Decomposing Systems into Modules*, CACM 12 (15), 1972] Przykład problemu: tekst, rotacje cykliczne wierszy. Możliwe podziały mogą organizować się albo wokół faz działania programu, albo wokół danych. Ten drugi sposób jest lepszy.

Ćwiczenia

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

- Sygnaturę półgrupy (typ elementów i operacja łączna).
- Sygnaturę monoidu (półgrupa z elementem neutralnym).
- Sygnatura grupy (monoid z elementami odwrotnymi).
- Sygnatura pierścienia (dodawanie, mnożenie, jedynka, zero, elementy przeciwne).
- Pierścień liczb całkowitych.
- Sygnatura ciała (pierścień z elementami odwrotnymi).
- Ciało liczb rzeczywistych.
- Sygnatura relacji binarnej na elementach tego samego zbioru (np. relacja równoważności, porządek). Przykładowe relacje.
- Sygnatura porządku z operacjami kresu górnego i dolnego. Porządek liniowy na liczbach całkowitych z operacjami `max` i `min`.

Wykład 12. Funktory

Funktory to moduły sparametryzowane. Często jeden moduł *importuje* inne moduły. Skoro rozdzieliliśmy pojęcia interfejsu (sygnatury) i implementacji (struktury), to możemy zapisywać moduły korzystające z innych modułów w bardziej abstrakcyjny sposób. Jeżeli spojrzymy na sygnaturę jak na specyfikację wykorzystywanego modułu, to zgodnie z zasadą ukrywania informacji *information hiding*, istotne powinny być jedynie specyfikacje wykorzystywanych modułów, a nie ich implementacje. Podstawienie dowolnej implementacji spełniającej zadaną specyfikację/sygnaturę powinno dać pożądane rezultaty. Moduł taki możemy zapisać w postaci sparametryzowanej — podajemy sygnatury parametrów, po podstawieniu w ich miejsce odpowiednich implementacji otrzymujemy wynikowy moduł.

Na funktory możemy też patrzeć jak na funkcje na strukturach — przetwarzają one moduły parametry w moduły wynikowe.

[[Składnia funktorów w BNFie.]]

Przykład: Kolejka priorytetowa. Na elementach kolejki musi być określony porządek liniowy.

```
type porownanie = Mniejsze | Rowne | Wieksze;;

module type PORZADEK_LINIOWY =
  sig
    type t
    val porownaj : t -> t -> porownanie
  end;;
```

Kolejka priorytetowa powinna mieć sygnaturę postaci:

```
module type PRI_QUEUE_FUNC = functor (Elem : PORZADEK_LINIOWY) ->
  sig
    exception EmptyQueue
    type elem = Elem.t
    type queue
    val empty : queue
    val put : elem -> queue -> queue
    val min : queue -> elem
    val removemin : queue -> queue
  end;;
```

Zwróćmy uwagę na to, że typ `queue` jest abstrakcyjny, a `elem` nie. Ukrywamy w ten sposób strukturę kolejki. Natomiast cokolwiek będzie wiadome o typie `Elem.t` będzie nadal wiadome o typie `elem`. Jest to konieczne z tego powodu, że inaczej nie bylibyśmy w stanie włożyć żadnego elementu do kolejki. Implementację kolejki priorytetowej możemy zrealizować w następujący sposób:

```
module PriQueue : PRI_QUEUE_FUNC =
  functor (Elem : PORZADEK_LINIOWY) ->
  struct
    type elem = Elem.t
```

```
    exception EmptyQueue
    type queue = elem list
    let empty = []
    let rec put x q =
      match q with
        [] -> [x] |
        h::t -> if Elem.porownaj x h = Mniejsze then
          x :: q
        else
          h::(put x t)
    let min q =
      match q with
        [] -> raise EmptyQueue |
        h::_ -> h
    let removemin (q:queue) =
      match q with
        [] -> raise EmptyQueue |
        _::t -> t
  end;;
```

Kolejka priorytetowa liczb całkowitych może być zrealizowana w taki sposób:

```
module IntOrder =
  struct
    type t = int
    let porownaj (x:int) (y:int) =
      if x < y then Mniejsze else
      if x > y then Wieksze else Rowne
  end;;

module IntQueue = PriQueue (IntOrder);;
```

W module `IntOrder` struktura typu `t` jest jawna. Dzięki temu będzie wiadomo, że elementy kolejki są typu `int`.

Innym podstawowym pojęciem związanym z porządkami liniowymi jest sortowanie.

```
module type SORTING = functor (Elem : PORZADEK_LINIOWY) ->
  sig
    type elem = Elem.t
    val sortuj : elem list -> elem list
  end;;
```

Jeden z algorytmów sortowania, to heap-sort.

```
module HeapSort (PrQ : PRI_QUEUE_FUNC): SORTING =
  functor (Elem : PORZADEK_LINIOWY) ->
    struct
      type elem = Elem.t
      module PQ = PrQ (Elem)
```

```
      open PQ
      let wkladaj l =
        fold_left (fun h x -> put x h) empty l
      let rec wyjmuj q a =
        if q = empty then
          rev a
        else
          wyjmuj (removemin q) (min q :: a)
      let sortuj l = wyjmuj (wkladaj l) []
    end;;

module Sortowanie = HeapSort (PriQueue);;

module S = Sortowanie (IntOrder);;

S.sortuj [1;7;3;6;5;2;4;8];;
```

Jak widać z powyższego przykładu, jeżeli będziemy zapisywać moduły w postaci funktorów, to będziemy z nich tworzyć łatwo wymienne elementy.

12.1 Funktory wyższych rzędów

Tak jak istnieją procedury wyższych rzędów, istnieją również funktory wyższych rzędów. Są to funktory, których parametrami i/lub wynikiem są funktory. Oto przykład, stanowiący kontynuację poprzedniego przykładu:

Przykład: Opierając się na sortowaniu możemy wprowadzić obliczanie mediany. Potrzebujemy do tego porządku liniowego i jakiegoś algorytmu sortowania.

```
module type MEDIANA =
  functor (Elem : PORZADEK_LINIOWY) ->
    sig
      type elem = Elem.t
      exception PustaLista
      val mediana : elem list -> elem
    end;;

module Mediana : functor (Sort : SORTING) -> MEDIANA =
  functor (Sort : SORTING) ->
    functor (Elem : PORZADEK_LINIOWY) ->
      struct
        type elem = Elem.t
        exception PustaLista
        module S = Sort (Elem)
        let mediana l =
          let s = S.sortuj l
          and n = List.length l
```

```
          in
            if n = 0 then
              raise PustaLista
            else
              List.nth s (n / 2)
      end;;

module M = Mediana (HeapSort (PriQueue)) (IntOrder);;

M.mediana [4;3;5;6;2;1;7];;
```

Przykład: Pomysł: Przestrzenie metryczne i grona przestrzeni. Funktory sklejające przestrzenie w grona oraz funktory określające sposoby łączenia gron w złożone przestrzenie metryczne. Uzupełnić.

Ćwiczenia

- Monoid składania funkcji — funktor, który dla określonego typu `t` konstruuje monoid z funkcją identycznościową i operacją składania funkcji (na `t`).
- Przypomnij sobie sygnaturę relacji binarnej z poprzednich ćwiczeń. Napisz funktor, który przekształca daną relację w jej domknięcie zwrotno-symetryczne.
- Podaj sygnaturę implementacji struktury danych liczb zespolonych (wyłącznie konstruktory i selektory biegunowe i prostokątne). Sygnatura powinna pasować do dowolnej implementacji: biegunowej, prostokątnej lub mieszanej. Naszkicuj implementację reszty pakietu liczb zespolonych, jako funktor, którego parametrem jest implementacja struktury danych, a wynikiem w pełni funkcjonalny pakiet.
- Przypomnijmy sobie pakiet liczb wymiernych z poprzedniego wykładu. Można go zapisać w postaci funktora, sparametryzowanego (wybranym typem) liczb całkowitych. Podaj sygnaturę liczb całkowitych zawierającą wszystkie operacje potrzebne do zaimplementowania pakietu liczb wymiernych ze skracaniem ułamków. (Potrzebne są: operacje arytmetyczne, równość, modulo i ew. porównywanie.) Naszkicuj budowę funktora implementującego taki pakiet liczb wymiernych.
- Wyobraźmy sobie pakiet implementujący wielomiany, wraz z rozmaitymi operacjami na wielomianach: suma, różnica, iloczyn, iloraz, reszta z dzielenia, porównanie, pochodna, itp. Pakiet taki może być sparametryzowany typem liczb (ciałem), nad którym rozpatrujemy wielomiany. Podaj odpowiednie sygnatury i naszkicuj sposób implementacji pakietu wielomianów jako funktora. Czy implementacja jest na tyle ogólna, aby pakiet działał nie tylko dla liczb zmiennopozycyjnych, ale również dla zespolonych i wymiernych?
- Korzystając z wyników poprzednich ćwiczeń podaj, jak można skonstruować pakiet funkcji wymiernych? (Jest to pakiet liczb wymiernych, których współczynniki są wielomianami.)
- Korzystając z wyników poprzednich ćwiczeń podaj, jak można skonstruować pakiet wielomianów dwóch zmiennych? (Są to wielomiany jednej zmiennej, których współczynniki są wielomianami drugiej zmiennej.)
- Porównaj funktory z mechanizmem dziedziczenia w programowaniu obiektowym. W jaki sposób można zasymulować hierarchię klas za pomocą funktorów? Co z rzutowaniem typów? Co z metodami wirtualnymi? (Nie wiem czy to dobre zadanie. Może tak, a może jest już zbyt późno, ale jakiś związek widzę.)

Wykład 13. Programowanie imperatywne

Paradygmat programowania imperatywnego opiera się na następującej analogii: Zwykle system obliczeniowy modeluje pewien fragment istniejącego świata rzeczywistego. W świecie rzeczywistym mamy do czynienia z obiektami, które w miarę upływu czasu zmieniają swoje stany. W systemie obliczeniowym możemy je modelować za pomocą obiektów obliczeniowych, które też mogą zmieniać stany.

13.1 Referencje

Programy imperatywne można pisać wykorzystując referencje:

- 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 polimorficzną możemy ją ukonkretnić. Przypisując wartość możemy podać wartość bardziej ogólną. Przypisując wartość konkretniejszą powodujemy ukonkretnienie typu referencji!

```
let f x y = x;;
val f : 'a -> 'b -> 'a = <fun>
let g x y = x + 1;;
val g : int -> 'a -> int = <fun>
let h x y = x + y;;
val h : int -> int -> int = <fun>
let i x y = if y then x else 2 * x;;
val i : int -> bool -> int = <fun>
let r = ref g;;
r := f;;
r := h;; -- zmienia się typ r!
r := i;; -- błąd
```

13.2 Konstrukcje imperatywne

W momencie, gdy mamy operacje imperatywne, istotne stają się ich efekty uboczne (a nie koniecznie wartość) oraz kolejność wykonania. Przydatne stają się następujące konstrukcje:

- średnik `;`,
- nawiasy `begin ...end`,
- tablice (poniżej),
- pętla for: `for v = wyr to wyr do ... done`,
- pętla while: `while wyr do ... done`,
- wyrażenie warunkowe postaci: `if` w `then` x,

- wypisywanie: `print_...`,
- wczytywanie: `read_...`.

13.3 Tablice

- `make n x` = nowa tablica rozmiaru `n` wypełniona wartościami `x`,
- `init n f` = nowa tablica rozmiaru `n` wypełniona kolejno wartościami `f i`,
- `[| ... |]` — konstruktor tablic,
- `length a` = długość tablicy,
- `a.(n)` = `get a n` = `n`-ty element tablicy `a`, elementy są numerowane od 0 do `length a - 1`,
- `a.(n) <- x` = `set a n x` = `unit` powoduje ustawienie wartości `a.(n)` na `x`.

13.4 Obiekty i stany

W jaki sposób możemy tworzyć obiekty obliczeniowe zdolne do zmiany stanu? Obiektami takimi są referencje. Można jednak tworzyć obiekty lokalne, dostępne tylko wybranym procedurom. Obiekt może być procedurą, wraz z wartościami lokalnymi widocznymi tylko dla niej. Może ona zmieniać te wartości za pomocą przypisania (`:=`).

```
let generator s =
  let r = ref s
  in
    function x -> begin
      r := !r + x;
      !r
    end;;
```

Zauważmy, że każde wywołanie procedury `generator` wiąże się z powstaniem osobnej ramki zawierającej argument `s`. Wynikiem każdego takiego wywołania jest procedura, która może zmieniać wartość zapamiętaną jako `s`.

```
let o = generator 0;;
o 22;;
- : int = 22
o 20;;
- : int = 42
```

13.5 Przykład: konto bankowe

Stan obiektu jest wyznaczony przez zawarte w nim *zmienne stanowe*. Dobrym zwyczajem jest wprowadzenie bariery abstrakcji oddzielającej zmienne stanowe obiektu od „świata zewnętrznego”. Obiekt powinien natomiast udostępniać metody umożliwiające zmianę jego stanu. Ustalamy więc *interfejs obiektu* złożony z metod zmiany stanu obiektu. Ukrywamy natomiast *implementację* stanu obiektu za pomocą jego zmiennych stanowych.

Możemy tutaj zastosować technikę *przesyłania komunikatów* — to obiekt sam zmienia swój stan na skutek otrzymania komunikatu opisującego zdarzenie powodujące zmianę.

Ponieważ zwykle operujemy na wielu obiektach, powinniśmy zastosować technikę *hermetyzacji*. Każdy obiekt powinniśmy zamknąć w osobnym „pudełku". Jedynie metody udostępniane przez obiekt mogą mieć dostęp do wnętrza obiektu. Dodatkowo mamy *konstruktor* — procedurę tworzącą nowe obiekty.

```
let konto pocz =
  let saldo = ref pocz
  in
    let wplata kwota =
      if kwota > 0 then begin
        saldo := !saldo + kwota;
        !saldo
      end else failwith "Ujemna lub zerowa kwota"
    and wyplata kwota =
      if kwota > 0 then
        if kwota <= !saldo then begin
          saldo := !saldo - kwota;
          !saldo
        end else failwith "Brak środków na koncie"
      else failwith "Ujemna lub zerowa kwota"
    in
      (wplata, wyplata);;

let (wplac, wyplac) = konto 0;;

wplac 50;;
wyplac 8;;
```

Co by się stało gdyby definicja `konto` nie miała argumentu `saldo`, lecz zaczynała się tak:

```
let konto =
  let saldo = ref 0
  in ...
```

13.6 Cena programowania imperatywnego

Konstrukcje imperatywne mają swoją cenę. To programowanie imperatywne wymusza np. wprowadzenie środowiskowego modelu obliczeń w takiej postaci, w jakiej został wcześniej przedstawiony. Ale na tym nie koniec.

Konstrukcje imperatywne powodują kłopoty z pojęciem *tożsamości*. Kiedy x i y to to samo? Czy wtedy, kiedy wartość x i y jest taka sama? Ale przecież wartości obiektów mogą się zmieniać. To może wtedy, gdy każda zmiana wartości x powoduje analogiczną zmianę y? To nie są łatwe pytania.

Musimy wypracować jakiś język do opisu tożsamości obiektów obliczeniowych. Będziemy mówić, że:

- (w danej chwili) x i y są równe, jeżeli ich wartości są równe, sprawdza to predykat `=`,

- obiekty x i y są tożsame, jeżeli są sobie równe i zmiana wartości jednego powoduje analogiczną zmianę drugiego; tożsamość sprowadza się do tego, że oba obiekty znajdują się w tym samym miejscu w pamięci, sprawdza to predykat `==`,

- obiekty x i y są niezależne, jeżeli zmiana stanu jednego z nich nie powoduje żadnej zmiany stanu drugiego z nich.

Cena imperatywności obejmuje też weryfikację programów imperatywnych i ich pielęgnację. Analiza programów imperatywnych jest bardzo trudna, ze względu na konieczność śledzenia zależności wynikających z kolejności wykonywania instrukcji — „czy ta instrukcja powinna być najpierw, czy tamta?" Ponadto, modyfikowanie programów imperatywnych, dodawanie nowych instrukcji, zmiana kolejności instrukcji, jest źródłem wielu błędów w programach, gdyż łatwo przeoczyć niektóre z zależności. Błędy te nie występują w programach funkcyjnych.

Cena ta rośnie jeszcze bardziej jeżeli dopuścimy *współbieżne* wykonanie kilku procedur operujących na tych samych obiektach. Wówczas bardzo trudno jest przewidzieć wszystkie możliwe scenariusze ich wykonania i przewidzieć wszystkie możliwe przeploty modyfikacji stanów obiektów. Przekonacie się o tym sami na *programowaniu współbieżnym*, che che ....

13.7 Funkcyjnie, czy imperatywnie?

Kiedy programować funkcyjnie, a kiedy imperatywnie? To że programowanie imperatywne ma swoją cenę nie oznacza, że wszystko i zawsze należy programować funkcyjnie. Tak jak dobry mechanik powinien znać wszystkie narzędzia i stosować je zgodnie z przeznaczeniem, tak samo dobry programista powinien znać wszystkie techniki programistyczne i paradygmaty programowania i wybierać te, które najlepiej nadają się do rozwiązania danego zadania. Nie wbijamy gwoździ kluczem francuskim i nie dokręcamy cieknącego kranu młotkiem!

Niektóre algorytmy (np. dynamiczne) dużo łatwiej jest zapisać z użyciem *imperatywnych struktur danych*, np. tablic. Nie przeszkadza to jednak takiemu ich zdefiniowaniu, aby cała imperatywność była ukryta przed użytkownikiem, stanowiła sekret implementacji.

Poniżej przedstawiono kolejny przykład mechanizmu, który można zaimplementować elegancko tylko wtedy, gdy jego serce będzie imperatywne.

Niektóre z kolei algorytmy łatwiej jest zapisać funkcyjnie. Dotyczy to zwłaszcza algorytmów operujących na procedurach wyższego rzędu, wszelkiego rodzaju drzewach i algorytmów rekurencyjnych. W takich przypadkach należy stosować programowanie funkcyjne.

13.8 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, sprawdzana własność i liczba prób. Wynikiem jest przybliżenie prawdopodobieństwa, z jakim dane mają własność.

```
let montecarlo dane wlasnosc liczba_prob =
  let rec montecarlo_rec a n =
    if n = 0 then
      a
    else
      if wlasnosc (dane ()) then
        montecarlo_rec (a+1) (n-1)
      else
        montecarlo_rec a (n-1)
  in
    float (montecarlo_rec 0 liczba_prob) /. float (liczba_prob);;
```

Spróbujmy za pomocą metody Monte Carlo przybliżyć wartość π. Liczba ta jest równa powierzchni koła jednostkowego.

```
let square x = x *. x;;
let kolo (x, y) = square x +. square y <= 1.0;;
let kwadrat () = (losowa_od_do (-1.0) 1.0, losowa_od_do (-1.0) 1.0);;
let pi = 4.0 *. (montecarlo kwadrat kolo 1000);;
```

Pozostał do zaprogramowania generator liczb losowych.

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

let losowa () =
  let duzo = 1000
  in
    let l = float (losowy_int () mod (duzo + 1)) /. float (duzo)
    in
      if l < 0.0 then (-. l) else l;;

let losowy_int =
  let stan = ref 0
  and a = 937126433
  and b = 937187
  in function () ->
    (
      stan := !stan * b + a;
      !stan
    );;
```

W tym przykładzie użyliśmy niewiele imperatywności, ale miała ona duży wpływ na strukturę programu. W przeciwnym przypadku musielibyśmy przekazywać stan generatora liczb losowych i to w wielu procedurach, przekraczając granice abstrakcji chroniące generator liczb losowych.

Laboratorium

Zadanie rozgrzewkowe, termin wykonania: 1 tydzień.

Tam gdzie dokonujemy pomiarów wielkości fizycznych, wyniki są obarczone pewnym błędem, np. 5m +/- 10%. Każdą taką przybliżoną wartość traktujemy jak zbiór możliwych wartości. Zaimplementuj pakiet operacji arytmetycznych na takich przybliżonych wartościach zawierający:

- konstruktory:
  - `wartosc_dokladnosc x p = x` $\pm$ `p%` (dla $p > 0$),
  - `wartosc_od_do x y = (x+y)/2` $\pm$ `(y-x)/2` (dla `x` $<$ `y`),
- selektory:
  - `val in_wartosc` x y $\Leftrightarrow$ wartość x może być równa y,
  - `min_wartosci x =` najmniejsza możliwa wartość `x`,
  - `max_wartosci x =` największa możliwa wartość `x`,
  - `sr_wartosci x =` średnia (arytmetyczna) wartość `x`,
- modyfikatory:
  - `plus a b` $= \{x + y : $ `in_wartosc a` $x \wedge$ `in_wartosc b` $y\}$,
  - `minus` $= \{x - y : $ `in_wartosc a` $x \wedge$ `in_wartosc b` $y\}$,
  - `razy` $= \{x \cdot y : $ `in_wartosc a` $x \wedge$ `in_wartosc b` $y\}$,
  - `podzielic` $= \{\frac{x}{y} : $ `in_wartosc a` $x \wedge$ `in_wartosc b` $y\}$.

Selektory, w przypadku, gdy wynik nie jest liczbą rzeczywistą, powinny zwracać odpowiednią z wartości: `infinity`, `neg_infinity` lub `nan`. Modyfikatory mogą domykać wynikowe zbiory wartości.

Całość powinna mieć postać modułu pasującego do następującej sygnatury:

```
module type ARYTMETYKA_PRZEDZIALOW = sig
  type wartosc
  val wartosc_dokladnosc: float -> float -> wartosc
  val wartosc_od_do: float -> float -> wartosc
  val wartosc_dokladna: float -> wartosc

  val in_wartosc: wartosc -> float -> bool
  val min_wartosci: wartosc -> float
  val max_wartosci: wartosc -> float
  val sr_wartosci:  wartosc -> float

  val plus: wartosc -> wartosc -> wartosc
  val minus: wartosc -> wartosc -> wartosc
  val razy: wartosc -> wartosc -> wartosc
  val podzielic: wartosc -> wartosc -> wartosc
end;;
```

Ćwiczenia

- Napisz moduł `Counter` o następującej sygnaturze:

```
module type COUNTER = sig
  type counter
  val make :  unit -> counter
  val inc :  counter -> int
  val reset :  unit -> unit
end;;
```

  Procedura `make` tworzy nowy licznik o początkowej wartości 0. Procedura `inc` zwiększa licznik o 1 i zwraca jego nową wartość. Procedura `reset` ustawia wartość **wszystkich** liczników na 0.

  Podaj złożoność czasową i pamięciową ciągu m operacji, spośród których n to operacje `make`.

- Drzewo binarne z fastrygą, to drzewo, w którym każdy węzeł posiada dodatkowy wskaźnik na następny węzeł w porządku infiksowym. (Ostatni węzeł powinien zawierać wskaźnik na korzeń.) Napisz procedurę `fastryguj`, która sfastryguje dane drzewo binarne.

```
type α drzewo =
  Puste |
  Węzeł of α drzewo * α drzewo * α drzewo ref;;
val fastryguj: α drzewo → unit;;
```

- Napisz procedurę sprawdzającą, czy dana lista zawiera cykl. Można to zrobić w stałej pamięci i liniowym czasie (i to na kilka sposobów)!

- Dana jest n-elementowa tablica znaków. Napisz procedurę `rotacja : char array -> int -> unit`, która rotuje daną tablicę o zadaną liczbę miejsc cyklicznie w prawo, tzn.:

$$\texttt{rotacja}[|x_1; x_2; \ldots; x_n|]\ k$$

  zmienia zawartość tablicy na:

$$[|x_{n-k+1}; \ldots; x_n; x_1; \ldots; x_{n-k}|]$$

  Na przykład, wywołanie:
  `rotacja [|'a'; 'l'; 'a'; 'm'; 'a'; 'k'; 'o'; 't'; 'a'|] 4`
  zmienia zawartość tablicy na `[|'k'; 'o'; 't'; 'a'; 'a'; 'l'; 'a'; 'm'; 'a'|]`.

- Dana jest tablica liczb całkowitych, której pewna rotacja jest posortowana ściśle rosnąco. Napisz procedurę `minmax :  int array -> int * int`, która znajduje minimum i maksimum w takiej tablicy.

Wykład 14. Imperatywne wskaźnikowe struktury danych

W poprzednim wykładzie przedstawione zostały imperatywne mechanizmy w języku programowania. W tym wykładzie zobaczymy na przykładach, jak mechanizmy te mogą zostać wykorzystane do implementacji efektywnych struktur danych.

14.1 Dwustronne kolejki ze scalaniem i odwracaniem

Przypuśćmy, że potrzebujemy zaimplementować dwustronne kolejki, z dodatkowymi operacjami: scalaniem i odwracaniem. Scalenie dwóch kolejek powoduje, że pierwsza z nich staje się wynikiem ich sklejenia, a druga staje się pusta. Odwracanie powoduje odwrócenie kierunku kolejki. Operacje te ujmuje następująca sygnatura:

```
module type QUEUE =
  sig
    type α queue
    exception EmptyQueue
    val init : unit → α queue
    val is_empty : α queue → bool
    val put_first : α queue → α → unit
    val put_last : α queue → α → unit
    val first : α queue → α
    val last : α queue → α
    val remove_first : α queue → unit
    val remove_last : α queue → unit
    val merge : α queue → α queue → unit
    val rev : α queue → unit
  end;;
```

14.1.1 Implementacja

- Technika atrap i strażników.
- Lista dwukierunkowa, bez określania kierunków dowiązań. Strażnicy na końcach. Lista, to para dowiązań do strażników.

  ```
  type α opt = Null | Val of α
  type α elem = { mutable l1 : α elem; mutable l2 : α elem; v : α opt }
  type α queue = { mutable front : α elem; mutable back : α elem }
  exception EmptyQueue
  ```

- Operacje:

  ```
  (* Wartość elementu. *)
  let value e =
    match e.v with
      Val x -> x |
  ```

```
    Null  -> raise EmptyQueue

(* Predykat sprawdzający czy kolejka jest pusta. *)
let is_empty q =
  (* Kolejka jest pusta gdy zawiera tylko strażników. *)
  let f = q.front
  and b = q.back
  in
    assert (not (f == b));
    assert (f.l1 == f);
    assert (b.l1 == b);
    assert ((f.l2 == b) = (b.l2 == f));
    (f.l2 == b)

(* Konstruktor pustej kolejki. *)
let init () =
  let rec g1 = { l1 = g1; l2 = g2; v = Null}
  and     g2 = { l1 = g2; l2 = g1; v = Null}
  in
    { front = g1; back = g2 }

(* Wstawia nowy element z wartością x pomiędzy elementy p i q. *)
let put_between p q x =
  let r = { l1 = p; l2 = q; v = Val x }
  in begin
assert (not (p == q) &&
        ((p.l1 == q) || (p.l2 == q)) &&
        ((q.l1 == p) || (q.l2 == p)));
    if p.l1 == q then p.l1 <- r else p.l2 <- r;
    if q.l1 == p then q.l1 <- r else q.l2 <- r
  end

(* Wstawienie na początek. *)
let put_first q x =
  let f = q.front
```

```
  in
    put_between f f.l2 x

(* Wstawienie na koniec. *)
let put_last q x =
  let b = q.back
  in
    put_between b b.l2 x

(* Pierwszy element. *)
let first q =
  value q.front.l2

(* Ostatni element. *)
let last q =
  value q.back.l2

(* Usuwa element (nie będący strażnikiem). *)
let remove e =
  assert (not (e.l1 == e) && (e.v <> Null));
  let e1 = e.l1
  and e2 = e.l2
  in begin
    assert (not (e1 == e2) && not (e1 == e) && not (e2 == e));
    if e1.l1 == e then e1.l1 <- e2 else e1.l2 <- e2;
    if e2.l1 == e then e2.l1 <- e1 else e2.l2 <- e1
  end

(* Usunięcie pierwszego el. *)
let remove_first q =
  if is_empty q then
    raise EmptyQueue
  else
    remove q.front.l2
```

```
(* Usunięcie ostatniego el. *)
let remove_last q =
  if is_empty q then
    raise EmptyQueue
  else
    remove q.back.l2

(* Sklejenie dwóch kolejek, druga kolejka ulega destrukcji. *)
let merge q1 q2 =
  assert (not (q1 == q2));
  let f1 = q1.front
  and b1 = q1.back
  and f2 = q2.front
  and b2 = q2.back
  in
    assert (not (f1==b1) && not (f2==b2));
    let e1 = b1.l2
    and e2 = f2.l2
    in begin
      (* Złączenie kolejek. *)
      if e1.l1 == b1 then e1.l1 <- e2 else e1.l2 <- e2;
      if e2.l1 == f2 then e2.l1 <- e1 else e2.l2 <- e1;
      (* Wynik w q1 *)
      q1.back <- b2;
      (* q2 puste *)
      f2.l2 <- b1;
      b1.l2 <- f2;
      q2.back <- b1
    end

(* Odwrócenie kolejki. *)
let rev q =
  let f = q.front
  and b = q.back
  in begin
```

```
      q.front <- b;
      q.back <- f
    end
```

W oczywisty sposób, złożoność wszystkich operacji na kolejkach jest rzędu $O(1)$.

14.2 Drzewa Find-Union

Wyobraźmy sobie, że potrzebujemy struktury danych do reprezentowania relacji równoważności oraz jej klas abstrakcji. Potrzebne są następujące typy i operacje:

- `type` α `set` — typ reprezentujący klasy abstrakcji elementów typu α,
- `make_set` : $\alpha \to \alpha$ `set` — tworzy jednoelementową klasę abstrakcji zawierającą x,
- `find` : α `set` $\to \alpha$ — zwraca ustalonego reprezentanta klasy abstrakcji,
- `equivalent` : α `set` $\to \alpha$ `set` $\to$ `bool` — sprawdza, czy dwie klasy abstrakcji są tą samą klasą abstrakcji,
- `union` : α `set` $\to \alpha$ `set` $\to$ `unit` — skleja dwie klasy abstrakcji, tworząc nową klasę abstrakcji, której reprezentantem jest reprezentant jednej ze sklejanych klas,
- `n_of_sets` : `unit` $\to$ `int` — zwraca liczbę rozłącznych klas abstrakcji,
- `elements` : α `set` $\to \alpha$ `list` — zwraca listę elementów tworzących klasę abstrakcji.

Powyższy typ i operacje można ująć w następującą sygnaturę:

```
module type FIND_UNION = sig
  type 'a set
  val make_set : 'a -> 'a set
  val find : 'a set -> 'a
  val equivalent : 'a set -> 'a set -> bool
  val union : 'a set -> 'a set -> unit
  val elements : 'a set -> 'a list
  val n_of_sets : unit-> int
end;;
```

Wartości typu α `set` reprezentują klasy abstrakcji zawierające wartości typu α, jednak dopuszczamy, aby różne (w sensie `==`) wartości typu α `set` reprezentowały tę samą klasę abstrakcji.

Każde wywołanie `make_set` tworzy odrębną, jednoelementową klasę abstrakcji. Jeżeli wszystkie wartości, na których wykonano operację `make_set` są różne, to można przyjąć, że dwa elementy x i y są w tej samej klasie abstrakcji wtw., gdy dla wygenerowanych dla nich klas abstrakcji c_x i c_y zachodzi `find` $c_x =$ `find` c_y. Jeśli tak nie jest, to argumenty `make_set` i wyniki `find` należy traktować jak etykiety właściwych elementów klas abstrakcji, które nie muszą być unikalne.

Zauważmy, że implementacja tak wyspecyfikowanej struktury danych **musi być imperatywna**. Wykonanie operacji `union` wpływa nie tylko na jej argumenty, ale również na wszystkie wartości (wygenerowane jako wyniki `make_set`) reprezentujące scalane klasy abstrakcji. Dodatkowo, efektywna implementacja wymaga zastosowania wskaźnikowej struktury danych.

Przykład: Efekty uboczne operacji `make_set`:

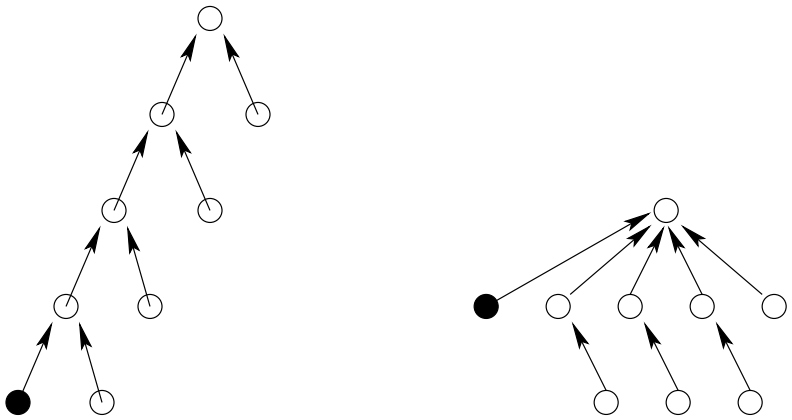
```
n_of_sets();;
- : int = 0
let a = make_set "a"
and b = make_set "b"
and c = make_set "c"
and d = make_set "d";;
n_of_sets();;
- : int = 4
union a b;;
union c d;;
n_of_sets();;
- : int = 2
a == d;;
- : bool = false
a == b;;
- : bool = false
equivalent a b;;
- : bool = true
equivalent a d;;
- : bool = false
union b c;;
equivalent a d;;
- : bool = true
```

14.2.1 Implementacja

Klasy abstrakcji będziemy implementować w postaci drzew, których węzły reprezentują elementy klas abstrakcji. Przy tym z każdego węzła będzie wychodzić wskaźnik do jego ojca. Reprezentantem klasy będzie korzeń drzewa. Wskaźnik wychodzący z korzenia prowadzi do niego samego. Wynikiem operacji `make_set` jest utworzenie wierzchołka, z którego wychodzi wskaźnik wskazujący na niego samego. Operacja `find` polega na przejściu wzdłuż wskaźników aż do korzenia drzewa i zwróceniu reprezentowanego przez niego elementu. Operacja `union` polega na znalezieniu korzeni dwóch drzew i podłączeniu jednego jako syna drugiego. Dodatkowo stosujemy dwie optymalizacje.

Kompresja ścieżek Za każdym razem, gdy wyszukujemy reprezentanta klasy, we wszystkich węzłach na ścieżce od danego węzła do korzenia przestawiamy wskaźniki na korzeń drzewa.

Wybór nowego korzenia W momencie gdy scalamy dwa drzewa, możemy wybrać korzeń którego z nich będzie korzeniem nowego drzewa. Generalnie, powinniśmy podłączać mniejsze drzewo do większego. Nie będziemy jednak pamiętać ani wielkości poddrzew zakorzenionych w poszczególnych wierzchołkach, ani ich wysokości. Zamiast tego, każdemu wierzchołkowi przypiszemy *rangę*, liczbę całkowitą, która jest górnym ograniczeniem na wysokość danego wierzchołka. W momencie tworzenia wierzchołka, jego ranga wynosi 0. Kompresja ścieżek nie



Rysunek 1: Kompresja ścieżek

wpływa na rangi wierzchołków. Gdy scalamy dwa drzewa, to podłączamy korzeń o mniejszej randze, do korzenia o większej randze. Jeżeli oba korzenie mają tę samą rangę, to wybieramy którykolwiek z nich i zwiększamy jego rangę o 1.

Wyliczanie elementów klas i licznik klas Opisana struktura danych pozwala na szybką implementację operacji `find` i `union`, ale nie pozwala na wyliczanie elementów klas. Potrzebujemy do tego wskaźników idących w dół drzewa. Dla każdej klasy abstrakcji utrzymujemy również drzewo „rozpinające" elementy klasy. Korzeniem tego drzewa jest reprezentant klasy abstrakcji. W każdym węźle drzewa pamiętamy listę jego następników. Kształty drzew tworzonych przez wskaźniki idące „w górę" i „w dół" nie muszą się pokrywać.

Dodatkowo, aby odpowiadać na pytania o liczbę klas abstrakcji, utrzymujemy licznik klas.

```
module Find_Union : FIND_UNION = struct
  (* Typ klas elementów typu 'a. *)
  type 'a set = {
    elem         : 'a;           (* Element klasy. *)
    up           : 'a set ref;   (* Przodek w drzewie find-union. *)
    mutable rank : int;          (* Ranga w drzewie find-union. *)
    mutable next : 'a set list   (* Lista potomków w pewnym drzewie
rozpinającym klasę. *)
  }

  (* Licznik klas. *)
  let sets_counter = ref 0

  (* Liczba wszystkich klas. *)
  let n_of_sets () = !sets_counter

  (* Tworzy nową klasę złożoną tylko z danego elementu. *)
  let make_set x =
    let rec v = { elem = x; up = ref v; rank = 0; next = [] }
```

```
  in begin
    sets_counter := !sets_counter + 1;
    v
  end

(* Znajduje korzeń drzewa, kompresując ścieżkę. *)
let rec go_up s =
  if s == !(s.up) then s
    else begin
      s.up := go_up !(s.up);
      !(s.up)
    end

(* Znajduje reprezentanta danej klasy. *)
let find s =
  (go_up s).elem

(* Sprawdza, czy dwa elementy są równoważne. *)
let equivalent s1 s2 =
  go_up s1 == go_up s2

(* Scala dwie dane (rozłączne) klasy. *)
let union x y =
  let fx = go_up x
  and fy = go_up y
  in
    if not (fx == fy) then begin
      if fy.rank > fx.rank then begin
        fx.up := fy;
        fy.next <- fx :: fy.next
      end else begin
        fy.up := fx;
        fx.next <- fy :: fx.next;
        if fx.rank = fy.rank then fx.rank <- fy.rank + 1
      end;
      sets_counter := !sets_counter - 1
    end

(* Lista elementów klasy. *)
let elements s =
  let acc = ref []
  in
    let rec traverse s1 =
      begin
        acc := s1.elem :: !acc;
        List.iter traverse s1.next
      end
```

```
      in begin
        traverse (go_up s);
        !acc
      end

end;;
```

14.2.2 Analiza kosztu

W niniejszym punkcie zajmiemy się analizą łącznego kosztu wykonania m operacji, spośród których n to operacje `make_set`. Na początek przeanalizujmy kilka własności rang.

Lemat 4. *Dla każdego węzła, który nie jest korzeniem, jego ranga jest mniejsza od rangi jego ojca.*

Dowód. Dowód przebiega indukcyjnie po historii obliczeń. Operacja `union` podłączając jeden korzeń do drugiego zapewnia, że ojciec ma większą rangę niż nowy syn. Kompresja ścieżek również zachowuje powyższą własność. □

Lemat 5. *Dla dowolnego drzewa, którego korzeń ma rangę r, liczba węzłów w drzewie wynosi przynajmniej 2^r.*

Dowód. Dowód przebiega indukcyjnie po historii obliczeń. Oczywiście jednoelementowe drzewa tworzone przez `make_set` spełniają lemat. Kompresja ścieżek nie zmienia liczby węzłów w drzewie, ani rangi korzenia. Jeżeli scalamy drzewa, których korzenie mają różne rangi, to oczywiście lemat jest zachowany. Jeżeli scalamy drzewa, których korzenie mają taką samą rangę r, to każde z tych drzew ma przynajmniej 2^r węzłów, a po ich scaleniu powstaje drzewo, którego korzeń ma rangę $r+1$ i które zawiera przynajmniej 2^{r+1} elementów. □

Fakt 4. *Dla dowolnej rangi r istnieje co najwyżej $\frac{n}{2^r}$ węzłów rangi r lub większej.*

Dowód. Ustalmy rangę r. Ranga wierzchołków rośnie w wyniku operacji `union`. Usuńmy z naszego ciągu instrukcji wszystkie operacje `union`, które prowadzą do powstania wierzchołków o randze większej niż r. W rezultacie, każdy wierzchołek, który oryginalnie miał rangę równą r lub większą, ma rangę równą r i jest korzeniem pewnego drzewa.

Z lematu 5 wynika, że każdy taki wierzchołek jest korzeniem drzewa zawierającego przynajmniej 2^r węzłów. Ponieważ różne drzewa są rozłączne, a wszystkich wierzchołków jest n, więc wierzchołków rangi r (lub większej) może być co najwyżej $\frac{n}{2^r}$. □

Fakt 5. *Rangi węzłów nie przekraczają $\lfloor \log_2 n \rfloor$.*

Dowód. Jest to natychmiastowy wniosek z faktu 4. □

Def. 1. Funkcja $\log_2^i x$ jest zdefiniowana standardowo, przy czym może ona być nieokreślona. Funkcja $\log^*$ jest zdefiniowana następująco:

$$\log^* x = \min\{i \geq 0 : \log_2^i x \leq 1\}$$

□

Przykładowo, $\log^* 4 = 2$, $\log^* 65000 = 4$.

Tw. 4. *Łączny koszt wykonania m operacji na drzewach find-union (bez operacji* `elements`*), spośród których n to operacje* `make_set` *wynosi $O(m \cdot \log^* n)$.*

Dowód. Zdefiniujmy pomocniczą funkcję B, odwrotną do funkcji $\log^*$:

$$B(j) = \begin{cases} \underbrace{2^{\cdot^{\cdot^{2}}}}_{j \text{ razy}} & \\ 2 & \text{jeśli } j \geq 1 \\ 1 & \text{jeśli } j = 0 \\ -1 & \text{jeśli } j = -1 \end{cases}$$

Wierzchołki dzielimy na *bloki*, ze względu na $\log^*$ od ich rang. Bloki numerujemy tymi wartościami. Ponieważ bloki mają numery od 0 do co najwyżej $\log^*(\log_2 n) = \log^* n - 1$, więc wszystkich bloków jest co najwyżej $\log^* n$. Blok numer j zawiera rangi od $B(j-1)+1$ do $B(j)$.

Jeżeli pominiemy koszt operacji `go_up`, to pozostały koszt operacji wynosi $O(m)$. Możemy więc skupić się na koszcie operacji `go_up`. Pojedyncza operacja `go_up` przebiega ścieżkę od danego węzła do syna korzenia: $(x_0, x_1, \ldots, x_l)$. Rangi wierzchołków na ścieżce rosną. Wierzchołki na ścieżce należące do tego samego bloku występują po kolei.

Koszt operacji `go_up` przypisujemy wierzchołkom na ścieżce, przy czym dzielimy go na koszt *związany ze ścieżką* i koszt *związany z blokami*. Koszt związany z blokami przypisujemy ostatnim wierzchołkom na ścieżce należącym do poszczególnych bloków, oraz synowi korzenia, x_{l-1}. Koszt związany z blokami każdej operacji `go_up` wynosi co najwyżej jeden plus liczba bloków, czyli $O(1 + \log^* n)$. Tak więc koszt związany z blokami wszystkich operacji wynosi $O(m \cdot \log^* n)$. Należy jeszcze pokazać, że łączny koszt związany ze ścieżkami jest również rzędu $O(m \cdot \log^* n)$.

Zauważmy, że jeżeli jakiemuś wierzchołkowi został raz przydzielony koszt za blok, to już więcej nie będzie mu przydzielony koszt za ścieżkę. Jeżeli natomiast wierzchołkowi został w danej operacji `go_up` przydzielony koszt za ścieżkę, to w wyniku tej operacji zmienił się ojciec tego wierzchołka i to na taki o wyższej randze. Jeżeli ten nowy ojciec należy do innego bloku, to dany wierzchołek już więcej nie będzie miał przypisanego kosztu za ścieżkę. Tak więc, jeżeli dany wierzchołek należy do bloku nr j, to może on mieć przypisany koszt za ścieżkę co najwyżej $B(j) - B(j-1) - 1$ razy.

Oznaczmy przez $N(j)$ liczbę wierzchołków należących do bloku j.

$$N(0) \leq \frac{n}{2^0} = \frac{n}{B(0)}$$

Dla $j > 0$ mamy:

$$N(j) \leq \frac{n}{2^{B(j-1)+1}} = \frac{n}{2B(j)} < \frac{n}{B(j)}$$

Łączną liczbę opłat za ścieżki możemy oszacować mnożąc, dla każdego bloku, maksymalną liczbę wierzchołków w tym bloku przez maksymalną liczbę opłat za ścieżkę.

$$\begin{aligned} \sum_{j=0}^{\log^* n - 1} (N(j) \cdot (B(j) - B(j-1) - 1)) &\leq \sum_{j=0}^{\log^* n - 1} \left(\frac{n}{B(j)} \cdot (B(j) - B(j-1) - 1) \right) \leq \\ &\leq \sum_{j=0}^{\log^* n - 1} \left(\frac{n}{B(j)} \cdot B(j) \right) = \sum_{j=0}^{\log^* n - 1} (n) \leq n \cdot \log^* n \end{aligned}$$

Tak więc łączny koszt wszystkich operacji jest rzędu $O(m \log^* n)$. □

W praktyce czynnik $\log^* n$ można przyjąć za stały, gdyż nie przekracza on 5 dla wszelkich praktycznych wielkości. Zauważmy, że $B(5) \geq 10^{19\,728}$, natomiast liczba wszystkich atomów we Wszechświecie jest szacowana na 10^{80}.

Powyższe twierdzenie można rozszerzyć również na operacje `elements`, jeżeli np. zapewnimy, że każdy element pojawia się na listach elementów klas tylko raz. Wówczas łączny czas wykonania tych operacji nie przekracza n.

Funkcja Ackermanna (zgodnie z jej sformułowaniem podanym przez Hermesa [Her65, Bra83]) jest zdefiniowana następująco:

$$A(0, i) = i + 1$$

$$A(i, 0) = A(i - 1, 1)$$

$$A(i, j) = A(i - 1, A(i, j - 1))$$

Można pokazać, że łączny koszt m operacji, spośród których n to operacje `make_set`, jest jeszcze mniejszy, mianowicie jest rzędu $O(m \cdot \alpha(m, n))$, gdzie $\alpha(m, n)$ jest funkcją „odwrotną" do funkcji Ackermanna:

$$\alpha(m, n) = \min \left\{ i \geq 1 : A\left(i, \left\lfloor \frac{m}{n} \right\rfloor\right) > \log_2 n \right\}$$

W praktyce czynnik $\alpha(m, n)$ można przyjąć za stały, gdyż nie przekracza on 4 dla wszelkich praktycznych wielkości.

Ćwiczenia

1. Zaimplementuj imperatywne listy jednokierunkowe wraz z operacjami:

   - `append` — modyfikuje daną listę przez doklejenie na jej końcu innej listy; lista doklejana jest niszczona,
   - `rev` – odwraca zadaną listę; dwukrotne odwrócenie powinno dawać listę *tożsamą* z początkową.

2. Napisz procedurę tworzącą cykliczną listę modyfikowalną.

3. Zaimplementuj imperatywną kolejkę FIFO.

4. Zaimplementuj kolejki FIFO+LIFO jako imperatywne listy dwukierunkowe.

5. [XII OI, Skarbonki] Mamy n skarbonek otwieranych kluczykami. Do każdej skarbonki jest inny kluczyk. Kluczyki do skarbonek są powrzucane do skarbonek. Żeby dostać się do zawartości skarbonki, skarbonkę można otworzyć kluczykiem (jeśli się go ma) lub ją rozbić.

   Skarbonki są ponumerowane od 0 do $n-1$. Na podstawie tablicy a (rozmiaru n), takiej, że $a[i]$ = numer skarbonki, w której znajduje się kluczyk do skarbonki numer i, oblicz minimalną liczbę skarbonek, które należy rozbić, tak aby dostać się do zawartości wszystkich skarbonek.

6. [X OI, Małpki]

   [[A może by to przenieść do wykładu jako przykład zastosowania find-union?]]

   Na drzewie wisi n małpek ponumerowanych od 1 do n. Małpka z nr 1 trzyma się gałęzi ogonkiem. Pozostałe małpki albo są trzymane przez inne małpki, albo trzymają się innych małpek, albo jedno i drugie równocześnie. Każda małpka ma dwie przednie łapki, każdą może trzymać co najwyżej jedną inną małpkę (za ogon). Rozpoczynając od chwili 0, co sekundę jedna z małpek puszcza jedną łapkę. W ten sposób niektóre małpki spadają na ziemię, gdzie dalej mogą puszczać łapki (czas spadania małpek jest pomijalnie mały).

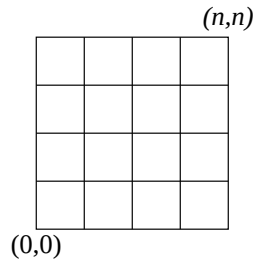
   Zaprojektuj algorytm, który na podstawie opisu tego która małpka trzyma którą, oraz na podstawie opisu łapek puszczanych w kolejnych chwilach, dla każdej małpki wyznaczy moment, kiedy spadnie ona na ziemię.

7. [XIV OI, Biura] W firmie pracuje n pracowników ponumerowanych od 1 do n. Każdy pracownik otrzymał służbowy telefon, w którym ma zapisane numery telefonów niektórych swoich współpracowników (a wszyscy ci współpracownicy mają w swoich telefonach zapisany jego numer). W związku z dynamicznym rozwojem firmy zarząd postanowił przenieść siedzibę firmy do nowych biurowców. Postanowiono, że każda para pracowników, którzy będą pracować w różnych budynkach, powinna znać (nawzajem) swoje numery telefonów, tzn. powinni oni mieć już zapisane nawzajem swoje numery w służbowych telefonach komórkowych. Równocześnie, zarząd postanowił zająć jak największą liczbę biurowców.

   Napisz procedurę `biura : int -> int * int list -> int`, która na podstawie liczby pracowników n oraz listy par pracowników posiadających nawzajem swoje numery telefonów $l = [(a_1, b_1); \ldots; (a_m, b_m)]$, wywołana jako `biura` n l, obliczy liczbę biurowców.

Rozwiązując to zadanie przyjmij, że $m = O(n)$.

8. Mamy daną kratownicę złożoną z przewodów elektrycznych wielkości $n \times n$.



   Niestety, poszczególne przewody przepalają się. Dana jest lista kolejnych jednostkowych przewodów, które się przepalają. Każdy przewód jest opisany trójką (x, y, v), gdzie (x, y) to współrzędna jego lewego/dolnego końca, a v jest wartością logiczną równą `true` dla przewodów pionowych i `false` dla przewodów poziomych.

   Napisz procedurę `przewody : int -> (int * int * bool) list -> int`, która dla danej liczby n i listy kolejnych przewodów przepalających się, określi po przepaleniu ilu przewodów prąd nie może płynąć między punktami $(0, 0)$ i (n, n). Jeżeli po przepaleniu się wszystkich przewodów z listy prąd nadal może płynąć, poprawnym wynikiem jest -1.

9. [ONTAK 2007] Wyobraźmy sobie ciąg zer i jedynek $(x_1, x_2, \ldots, x_n)$. Mamy dany ciąg trójek postaci (i, j, s), gdzie i i j są indeksami w ciągu (x_i), $1 \leq i \leq j \leq n$, natomiast $s = (\sum_{k=i}^{j} x_k) \bmod 2$. Niestety dany ciąg trójek jest sfałszowany — nie istnieje ciąg (x_i), który by go spełniał. Twoim zadaniem jest znalezienie takiego k, że dla pierwszych k danych trójek istniejej jeszcze spełniający je ciąg, a dla $k+1$ już nie. Napisz procedurę `przedzialy` : `int` $\rightarrow$ (`int` $*$ `int` $*$ `int`) `list` $\rightarrow$ `int`, która dla danego n oraz ciągu trójek oblicza takie k.

Wytyczne dla prowadzących ćwiczenia

Ad. 5 To zadanie można rozwiązywać zarówno stosując przeszukiwanie grafów, jak i drzewa find-union. (Wynik jest równy liczbie spójnych składowych.) Ponieważ o przeszukiwaniu grafów nic jeszcze nie było, należy zastosować drzewa find-union.

Ad. 7 Wynik jest równy liczbie spójnych składowych w dopełnieniu danego grafu. Nie należy jednak konstruować takiego dopełnienia, gdyż byłby to koszt czasowy rzędu $O(n^2)$. Należy znaleźć wierzchołek o minimalnym stopniu (jest on ograniczony przez stałą) i skleić z nim wszystkie wierzchołki, które nie są z nim incydentne. W ten sposób uzyskujemy graf stałej wielkości, który dalej przetwarzamy jakkolwiek, choćby w czasie $O(n^2)$.

Ad. 9 Oznaczmy przez p_i sumę (modulo 2) prefiksu ciągu (x_i) złożonego z pierwszych i elementów, $p_i = (\sum_k = 1^i x_k) \bmod 2$. W szczególności $p_0 = 0$. Trójka (i, j, s) oznacza, że $p_j = (p_{i-1} + s) \bmod 2$.

Dla każdego $i = 0, 1, \ldots, n$ mamy dwa wierzchołki, jeden reprezentujący fakt, że $p_i = 0$ i jeden reprezentujący fakt, że $p_i = 1$. Każda trójka to przyczynek do sklejenia ze sobą dwóch par wierzchołków, które muszą być sobie równoważne. Jeżeli po którymś z kolei sklejeniu wierzchołki $p_j = 0$ i $p_j = 1$ będą w tej samej klasie abstrakcji, to nie istnieje ciąg (x_i) spełniający przetworzone trójki.

Wykład 15. Logika Hoare'a — dowodzenie poprawności programów imperatywnych

W tym wykładzie zajmiemy się specyfikacjami i dowodzeniem poprawności programów imperatywnych. Zacznijmy od kilku podstawowych pojęć.

15.1 Kilka podstawowych pojęć

Większość z opisanych poniżej pojęć poznaliśmy już w odniesieniu do programów funkcyjnych. Określmy teraz co oznaczają one w odniesieniu do programów imperatywnych. (Jednocześnie powinno to wyjaśnić ich nazwy.)

- W przypadku programów imperatywnych istotne jest w jaki sposób zmieniają one **stan** systemu, czyli wartości zmiennych.
- **Asercja** to warunek odnoszący się do stanu w danym momencie obliczeń / w określonym miejscu w programie, który powinien być spełniony.
- **Warunek końcowy** to asercja odnosząca się do stanu po wykonaniu danego programu. Warunek końcowy określa stany systemu, do jakich powinien prowadzić program.
- **Warunek początkowy** to asercja odnosząca się do stanu przed wykonaniem danego programu. Warunek początkowy wyraża założenia, jakie powinien spełniać stan, w którym uruchamiamy program.
- Na potrzeby niniejszego wykładu przyjmujemy, że **specyfikacja** danego programu składa się z warunku początkowego i końcowego.
- Program jest **częściowo poprawny** (względem danej specyfikacji), jeżeli dla każdego stanu spełniającego warunek początkowy, program uruchomiony w tym stanie, o ile kończy działanie (bez błędu), to kończy je w stanie spełniającym warunek końcowy.
- Program ma **własność stopu**, jeżeli dla każdego stanu spełniającego warunek początkowy, program uruchomiony w tym stanie kończy działanie (bez błędu).
- Program jest **całkowicie poprawny** (względem danej specyfikacji), jeżeli dla każdego stanu spełniającego warunek początkowy, program uruchomiony w tym stanie kończy działanie i to w stanie spełniającym warunek końcowy. Inaczej mówiąc:

$$\text{Całkowita poprawność} = \text{Częściowa poprawność} + \text{własność stopu}$$

15.2 Formalizmy do specyfikacji i weryfikacji programów imperatywnych

Powstało wiele formalizmów służących do dowodzenia poprawności programów imperatywnych, m.in.:

- logika Hoare'a [Hoa69, Apt81, BA05, Dah92] — którą zajmiemy się w tym wykładzie,
- transformatory predykatów Dijkstry [Dij85, BA05] — służą one do wyznaczania najsłabszego warunku wstępnego, gwarantującego zakończenie obliczeń i spełnienie warunku końcowego,

- logika algorytmiczna [MS92, MS87] — w dużym skrócie, logika algorytmiczna stanowi rozszerzenie języka predykatów pierwszego rzędu o formuły postaci: program, warunek końcowy; taka formuła reprezentuje najsłabszy warunek wstępny gwarantujący zakończenie programu i spełnienie warunku końcowego; co więcej, takie formuły mogą być fragmentami większych formuł.

15.3 While-programy

W punkcie tym przedstawimy prosty fragment imperatywnego Ocamla, który będziemy nazywać *while-programami*. Dalsze rozważania ograniczymy do while-programów.

Dla uproszczenia zakładamy, że wszystkie „zmienne" są referencjami do liczb całkowitych, oraz że każdy identyfikator reprezentuje inną referencję. Rozszerzenie do referencji do innych typów prostych nie jest trudne, ale komplikuje technikalia. Natomiast wprowadzenie aliasingu, lub imperatywnych typów złożonych, takich jak tablice czy struktury wskaźnikowe, istotnie komplikuje problem specyfikacji i weryfikacji programów [Apt81, CO81, BS95, Kub00, Kub03].

```
⟨while-program⟩ ::= ⟨instrukcja⟩ | ⟨instrukcja⟩ ; ⟨while-program⟩
⟨instrukcja⟩    ::= begin ⟨while-program⟩ end |
                    if ⟨wyrażenie⟩ then ⟨instrukcja⟩ else ⟨instrukcja⟩ |
                    while ⟨warunek⟩ do ⟨instrukcja⟩ |
                    ⟨referencja⟩ := ⟨wyrażenie⟩ |
                    ()
⟨wyrażenie⟩     ::= ...
⟨warunek⟩       ::= ...
⟨identyfikator⟩ ::= ...
```

[[Dodefiniować składnię wyrażeń, warunków i referencji.]]

15.4 Semantyka while-programów

Skoro wszystkie zmienne są typu `int`, to stan systemu możemy reprezentować jako wartościowanie zmiennych, czyli funkcję z identyfikatorów w liczby całkowite.

$$Stan = [Id \to \mathbb{Z}]$$

Semantykę while-programu możemy reprezentować jako funkcję częściową ze stanów w stany. Częściowość wynika stąd, że obliczenia mogą się zapętlać, lub nie udawać z innych przyczyn.

$$[\![\text{while-program}]\!] : Stan \mapsto Stan$$

[[Rozwinąć równania]]

15.5 Formuły logiki Hoare'a

Formuły Hoare'owskie to trójki postaci:

$$\{\varphi\}P\{\psi\}$$

gdzie:

- φ to warunek początkowy,
- P to while-program,
- ψ to warunek końcowy.

Formuła taka wyraża częściową poprawność programu P względem φ i ψ.

15.6 System dowodzenia

- Aksjomaty:
  - instrukcji pustej:
    $$A \vdash \{\varphi\}()\{\varphi\}$$
  - instrukcji przypisania
    $$A \vdash \{\varphi[!\mathtt{x}/w]\}\mathtt{x} := w\{\varphi\}$$
    o ile warunek początkowy implikuje, że obliczenie w kończy się powodzeniem.

  Reguły dla:
  - osłabiania.
    $$\frac{A \models \varphi \Rightarrow \varphi', A \vdash \{\varphi'\}\mathtt{P}\{\psi'\}, A \models \psi' \Rightarrow \psi}{A \vdash \{\varphi\}\mathtt{P}\{\psi\}}$$
  - średnika:
    $$\frac{A \vdash \{\varphi\}\mathtt{P}\{\xi\}, A \vdash \{\xi\}\mathtt{Q}\{\psi\}}{A \vdash \{\varphi\}\mathtt{P};\mathtt{Q}\{\psi\}}$$
  - instrukcji warunkowej:
    $$\frac{A \models \varphi \Rightarrow (\alpha \vee \neg\alpha), A \vdash \{\varphi \wedge \alpha\}\mathtt{P}\{\psi\}, A \vdash \{\varphi \wedge \neg\alpha\}\mathtt{Q}\{\psi\}}{A \vdash \{\varphi\}\mathtt{if}\ \alpha\ \mathtt{then\ P\ else\ Q}\{\psi\}}$$
  - pętli while (wersja słaba, klasyczna):
    $$\frac{A \models \varphi \Rightarrow (\alpha \vee \neg\alpha), A \vdash \{\varphi \wedge \alpha\}\mathtt{P}\{\varphi\}}{A \vdash \{\varphi\}\mathtt{while}\ \alpha\ \mathtt{do\ P\ done}\{\varphi \wedge \neg\alpha\}}$$
  - pętli while (wersja silna, ale formalnie nie należy do tej logiki):
    $$\frac{\begin{array}{c}A \models \varphi \Rightarrow (\alpha \vee \neg\alpha), A \models (\varphi \wedge \mu \leq 0) \Rightarrow \neg\alpha, \\ n \text{ nie występuje w } \mathtt{P}, A \vdash \{\varphi \wedge \alpha \wedge \mu = n\}\mathtt{P}\{\varphi \wedge \mu < n\}\end{array}}{A \vdash \{\varphi\}\mathtt{while}\ \alpha\ \mathtt{do\ P\ done}\{\varphi \wedge \neg\alpha\}}$$

Przykład: Obliczanie minimum.

Przykład: Algorytm mnożenia rosyjskich chłopów.

Przykład: Algorytm Euklidesa.

Ćwiczenia

- silnia,
- liczby Fibonacciego,
- $\left(\begin{array}{c} n \\ k \end{array} \right)$
- Wyszukanie minimum w ciągu $(f(0), \ldots, f(n))$.

Wykład 16. Back-tracking

W tym wykładzie przedstawimy technikę rozwiązywania problemów nazywaną *przeszukiwaniem z nawrotami* lub z angielska *back-tracking*. Technika ta służy do rozwiązywania problemów, do których można zastosować zasadę „dziel i zwyciężaj", ale dany problem sprowadza się do podproblemu(ów) na wiele sposobów. Szukając więc rozwiązania musimy podjąć szereg decyzji jak podzielić/zmniejszyć rozpatrywany problem. Przeszukiwanie z nawrotami polega na rekurencyjnym przeszukiwaniu wszystkich możliwych ciągów takich decyzji.

Techniki tej możemy również użyć do rozwiązywania problemów optymalizacyjnych, czyli takich, gdzie nie tylko chcemy znaleźć możliwe rozwiązanie, ale rozwiązanie pod pewnym względem najlepsze. Jeżeli w trakcie przeszukiwania okazuje się, że wcześniej podjęte decyzje nie prowadzą do rozwiązania, lub nie prowadzą do rozwiązania lepszego od już znanego, to z takiej gałęzi rekurencyjnych przeszukiwań można się wycofać i spróbować innej gałęzi.

16.1 Prosty generyczny back-tracking

Zdefiniujemy teraz prosty ogólny mechanizm back-trackingu, dla problemów, w których dokonując wyboru sprowadzamy problem do jednego problemu o mniejszym rozmiarze. Tak więc szukane rozwiązanie możemy traktować jak ciąg prowadzących do niego wyborów.

Zdefiniujemy funktor, który na podstawie instancji problemu znajduje jedno lub wszystkie jego rozwiązania. Instancja problemu musi definiować następujące pojęcia:

- **konfiguracja** — jest to typ, którego wartości reprezentują dokonane wybory; kolejny wybór polega na zmianie konfiguracji,
- konfiguracja **końcowa** — to taka konfiguracja, która zawiera pełne rozwiązanie problemu,
- **wynik** — jest to typ reprezentujący rozwiązania; może on być uboższy niż typ konfiguracji; potrzebna jest również operacja **rzutowania** konfiguracji na wynik.
- dla danej konfiguracji określamy **iterator**, który dla danej procedury i konfiguracji, wywołuje tę procedurę dla wszystkich konfiguracji osiągalnych w jednym kroku.

Instancję problemu możemy opisać za pomocą modułów pasujących do następującej sygnatury:

```
module type BT_PROBLEM =
  sig
    (* Typ rozpatrywanych konfiguracji. *)
    type config

    (* Typ szukanych wyników. *)
    type result

    (* Czy dana konfiguracja jest już kompletnym rozwiązaniem. *)
    val final : config -> bool

    (* Wyciąga z konfiguracji istotne rozwiązanie.           *)
    (* W przypadu imperatywnych struktur danych tworzy kopię. *)
```

```
    val extract : config -> result

    (* Procedura, która wywołuje daną procedurę,          *)
    (* dla każdej konfiguracji osiągalnej w jednym kroku. *)
    val iter : (config -> unit) -> config -> unit
  end;;
```

Mechanizm znajdujący jedno rozwiązanie lub wszystkie rozwiązania wygląda następująco:

```
module type BT_SOLVER = functor (Problem : BT_PROBLEM) ->
  sig
    (* Wyjątek podnoszony, gdy brak rozwiązań. *)
    exception NoSolution

    (* Procedura znajdująca jedno rozwiązanie. *)
    val onesolution : Problem.config -> Problem.result

    (* Procedura znajdująca wszystkie rozwiązania. *)
    val solutions : Problem.config -> Problem.result list
  end;;

(* Implementacja *)
module Bt_Solver : BT_SOLVER = functor (Problem : BT_PROBLEM) ->
  struct
    (* Wyjątek podnoszony, gdy brak rozwiązań. *)
    exception NoSolution

    (* Wyjątek podnoszony, gdy znaleziono rozwiązanie. *)
    exception Solution of Problem.result

    (* Procedura znajdująca rozwiązanie. *)
    let onesolution s =
      let rec backtrack s =
        begin
          if Problem.final s then
            raise (Solution (Problem.extract s));
          Problem.iter backtrack s
        end
      in
        try (backtrack s; raise NoSolution)
        with Solution x -> x

    (* Procedura znajdująca wszystkie rozwiązania. *)
    let solutions s =
      let wynik = ref []
      in
        let rec backtrack s =
```

```
        begin
          if Problem.final s then
            wynik := (Problem.extract s)::!wynik;
          Problem.iter backtrack s
        end
      in begin
        backtrack s;
       !wynik
      end
end;;
```

Przedstawione mechanizmy nie uwzględniają problemów optymalizacyjnych, ani kryteriów pozwalających obcinać gałęzie przeszukiwania na podstawie znalezionych już rozwiązań. Nie pasuje on również do problemów, w których podjęty wybór sprowadza dany problem do wielu mniejszych problemów.

16.2 Problem ośmiu hetmanów

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 — pierwszy znaleziony.

Problem ośmiu hetmanów możemy opisać w postaci następującego modułu:

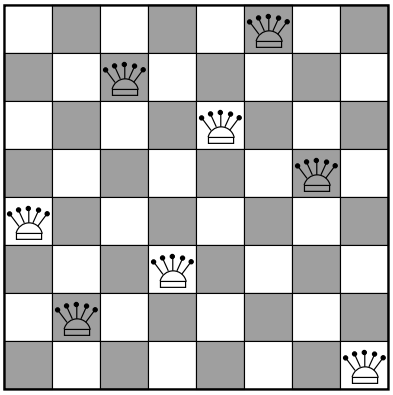
```
module Hetman =
  struct
    (* Pozycje hetmanów na szachownicy. *)
    type result = (int * int) list

    (* Typ konfiguracji na szachownicy.                      *)
    (* Trójka: wielkość planszy, liczba początkowych kolumn, *)
    (* w których należy postawić hetmany, lista ich pozycji. *)
    type config = int * int * result

    (* Pusta plansza rozmiaru n. *)
    let empty n = (n, n, [])

    (* Pozycje hetmanów w konfiguracji. *)
    let extract (_, _, l) = l

    (* Czy gotowe rozwiązanie. *)
    let final (_, k, _) = k=0

    (* Funkcja określająca, czy dwa hetmany się 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 mozna_dostawic h het =
      fold_left (fun ok x -> ok && not (szach h x)) true het

    (* Lista kolejnych liczb całkowitych od-do. *)
    let rec ints a b = if a > b then [] else a :: ints (a+1) b;;

    (* Konfiguracje powstałe przez dostawienie hetmana w kolumnie k. *)
    let iter p (n, k, l) =
      let r = filter (fun i -> mozna_dostawic (k, i) l) (ints 1 n)
      in List.iter (fun i -> p (n, k-1, (k,i)::l)) r
  end;;
```

Hetmany są ustawiane w kolejnych kolumnach od prawej do lewej. Rozwiązanie jest gotowe, gdy w każdej kolumnie stoi hetman. Kolejne konfiguracje powstają w wyniku ustawienia hetmana w ostatniej pustej kolumnie, we wszystkich miejscach, których nie szachują hetmany stojące już na planszy. Zwróćmy uwagę, że przerywamy sprawdzanie, jak tylko okaże się, że hetmany się szachują.

Dysponując modułem `Hetman`, możemy zastosować do niego funktor `SBt_Solver`.

```
(* Instancja problemu. *)
module H = Bt_Solver (Hetman);;

(* Procedura znajdująca pewne ustawienie hetmanów. *)
let hetman n = H.onesolution (Hetman.empty n);;

(* Procedura znajdująca ustawienie hetmanów. *)
let hetmany n = H.solutions (Hetman.empty n);;

hetman  4;;
- : Hetman.result = [(1, 3); (2, 1); (3, 4); (4, 2)]

hetmany 4;;
- : Hetman.result list =
  [[(1, 2); (2, 4); (3, 1); (4, 3)];
  [(1, 3); (2, 1); (3, 4); (4, 2)]]

hetman 8;;
- : Hetman.result =
  [(1, 4); (2, 2); (3, 7); (4, 3); (5, 6); (6, 8); (7, 5); (8, 1)]

hetmany 8;;
- : Hetman.result list =
  [[(1, 5); (2, 7); (3, 2); (4, 6); (5, 3); (6, 1); (7, 4); (8, 8)];
   [(1, 4); (2, 7); (3, 5); (4, 2); (5, 6); (6, 1); (7, 3); (8, 8)];
   [(1, 6); (2, 4); (3, 7); (4, 1); (5, 3); (6, 5); (7, 2); (8, 8)];
   [(1, 6); (2, 3); (3, 5); (4, 7); (5, 1); (6, 4); (7, 2); (8, 8)];
   [(1, 4); (2, 2); (3, 8); (4, 6); (5, 1); (6, 3); (7, 5); (8, 7)];
   [(1, 5); (2, 3); (3, 1); (4, ...); ...]; ...]
```



Rysunek 2: Przykładowe ustawienie ośmiu hetmanów na szachownicy.

16.3 Przykład: Sudoku

Łamigłówkę Sudoku definiujemy w następujący sposób: Plansza do Sudoku to kwadrat $n \times n$ złożony z mniejszych kwadratów $n \times n$. W rezultacie, jest to kwadrat złożony z $n^2 \times n^2$ pól. Sudoku należy wypełnić liczbami od 1 do n^2, tak żeby w każdej kolumnie, każdym wierszu, a także każdym mniejszym kwadracie każda liczba występowała dokładnie raz. Część liczb jest już na swoich miejscach, pozostałe trzeba odgadnąć.

4						6		5
		7	9	6			1	
	6		4			8		7
	8	3			6			
	1						5	
			2			4	6	
7		6			9		2	
	5			2	4	7		
8		9						6

Rysunek 3: Przykładowe sudoku dla $n = 3$.

Sudoku reprezentujemy jako dwuwymiarową tablicę liczb, którą będziemy stopniowo wypełniać. Puste miejsca są wypełnione zerami.

```
module Sudoku_Framework =
  struct
  open List

  (* Tablicowa reprezentacja łamigłówki.              *)
  (* Tablica n^2 x n^2 wypełniona liczbami 0..n^2.    *)
  (* Puste pola są reprezentowane przez zera.         *)
  type sudoku = int array array
```

```
(* Rozmiar n danego sudoku. *)
let size (s:sudoku) = truncate(sqrt (float (Array.length s)))

(* Lista kolejnych liczb całkowitych od-do. *)
let rec ints a b = if a > b then [] else a :: ints (a+1) b

(* Przekształca parę list w listę par - odpowiednik produktu. *)
let pairs l1 l2 =
  flatten (map (fun i-> map (fun j -> (i,j)) l1) l2)

(* Lista kolejnych indeksów sudoku rozmiaru n. *)
let indeksy s =
  let n = size s
  in ints 0 (n * n - 1)

(* Lista cyfr, którymi wypełniamy sudoku. *)
let cyfry s =
  let n = size s
  in ints 1 (n * n)

(* Lista par indeksów reprezentujących jeden "kwadrat". *)
let kwadrat s =
  let n = size s
  in
    let s = ints 0 (n-1)
    in pairs s s

(* Tworzy listę indeksów do pustych miejsc w sudoku. *)
let brakujace (s : sudoku) =
  filter
    (fun (i,j) -> s.(i).(j) = 0)
    (pairs (indeksy s) (indeksy s))
      ⋮
```

W dane puste pole możemy wstawić tylko taką liczbę, która nie występuje w danej kolumnie, wierszu i mniejszym kwadracie.

```
      ⋮
(* Sprawdza, czy cyfrę k można wstawić w miejscu (i,j). *)
let valid_value (s : sudoku) i j k =
  let valid_column i =
    for_all (fun x -> s.(i).(x) <> k || x = j) (indeksy s)
  and valid_row j =
    for_all (fun x -> s.(x).(j) <> k || x = i) (indeksy s)
  and valid_square i j =
    let n = size s
```

```
      in
        for_all
          (fun (x, y) -> (x, y) = (i,j) || s.(x).(y) <> k)
          (map (fun (x,y) -> (n*(i/n) + x, n*(j/n) + y)) (kwadrat s))
    in
      (valid_column i) &&
      (valid_row j) &&
      (valid_square i j)

  (* Wyznacza listę cyfr, które można wstawić w miejscu (i,j). *)
  let valid_values (s : sudoku) i j =
    filter (valid_value s i j) (cyfry s)
end;;
```

Rozwiązując sudoku możemy połączyć back-tracking z wnioskowaniem. Jeżeli na podstawie wypełnionych pól potrafimy wypełnić pewne pola, to należy tak zrobić. Jeśli nie potrafimy nic wywnioskować, należy za pomocą back-trackingu przejrzeć możliwe wartości wybranego pola.

Taką koncepcję możemy zaimplementować za pomocą funktora. Parametrem funktora jest procedura wnioskująca o niektórych polach. Wywnioskowane wartości wstawia ona do tablicy. Jej wynikiem są listy: wypełnionych w wyniku wnioskowania pól oraz pól, które nadal pozostają puste.

```
(* Typ modułu implementującego wnioskowanie n.t. sudoku. *)
module type SUDOKU_REASONING =
  sig
    (* Konfiguracja, to plansza i lista pustych pól. *)
    type config =  Sudoku_Framework.sudoku * (int * int) list

    (* Wynikiem reason jest lista pól wywnioskowanych i pozostałych pustych.
*)
    val reason : config -> (int * int) list * (int * int) list
  end;;
```

Funktor na podstawie procedury wnioskowania tworzy instancję problemu, który jest rozwiązywany za pomocą generycznego back-trackingu. Dodatkowo, w back-trackingu wybieramy do wypełnienia pole, na którym może się znajdować najmniej możliwych liczb.

```
(* Funktor, który wplata procedurę wnioskowania o sudoku *)
(* w back-tracking znajdujący rozwiązania.                *)
module Sudoku_BT (R : SUDOKU_REASONING) =
  struct
    open Sudoku_Framework

    (* Instancja problemu dla back-trackingu. *)
    module Problem =
      struct
        (* Wynikiem jest plansza sudoku. *)
```

```
type result = sudoku

(* Konfiguracja, to plansza i lista pustych pól. *)
type config = R.config

(* Sudoku jest skończone, gdy nie ma wolnych pól. *)
let final (s, l) = l = []

(* Kopia planszy. *)
let copy_sudoku s =
  Array.init (Array.length s) (fun i-> Array.copy s.(i))

(* Extract tworzy kopię planszy. *)
let extract (s, l) = copy_sudoku s

(* Procedura przeglądająca konfiguracje osiągalne w jednym kroku. *)
(* Wypełniane jest pierwsze pole z listy pustych pól.             *)
let iter p (s, l) =
  if not (final (s, l)) then
    let (wst, l1) = R.reason (s, l)
    in begin
      if not (final (s, l1)) then begin
        (* Wybierz z listy pozycji do wypełnienia tę o najmniejszej *)
        (* liczbie możliwych miejsc do wstawienia.                  *)

        (* Lista pozycji i możliwych ruchów, posortowana. *)
        let lr =
          let cmp (x, _, _) (y, _, _) =
            if x < y then -1 else if x > y then 1 else 0
          in
            List.sort cmp
              (List.map
                (fun (i, j) ->
                  let v = valid_values s i j
                  in (List.length v, (i, j), v))
                l1)
        in
          (* Pole z najkrotsza listą możliwości. *)
          let (_, (i, j), v) = List.hd lr
          (* Pozostałe pola. *)
          and t = List.map (fun (_, p, _) -> p) (List.tl lr)
          in
            List.iter
              (fun k ->
                begin
                  s.(i).(j) <- k;
                  p (s, t);
```

```
                        s.(i).(j) <- 0
                      end)
                  v
            end else p (s, l1);
            List.iter (fun (i,j) -> s.(i).(j) <- 0) wst
          end
    end

  (* Instancja rozwiązania za pomocą back-trackingu. *)
  module Solution = Bt_Solver (Problem)

  (* Procedura znajdująca rozwiązania dla danej planszy sudoku. *)
  let sudoku s =
    Solution.solutions (s, brakujace s)

end;;
```

To, że w pierwszej kolejności wypełniamy pole, dla którego liczba możliwych cyfr jest minimalna realizuje dwie dodatkowe strategie optymalizacyjne. Po pierwsze, jeżeli na planszy jest pole, na którym nie może znajdować się żadna cyfra, czyli sudoku nie da się rozwiązać, ucinamy wszelkie wywołania rekurencyjne dla tej konfiguracji. Jeżeli na planszy znajduje się pole, na którym może znajdować się tylko jedna cyfra, to w pierwszej kolejności wstawiana jest ta właśnie cyfra.

Prosty back-tracking możemy uzyskać nie stosując żadnego wnioskowania:

```
(* Brak wnioskowania o sudoku. *)
module NoReasoning =
  struct
    (* Konfiguracja, to plansza i lista pustych pól. *)
    type config =  Sudoku_Framework.sudoku * (int * int) list

    (* Nic nie wnioskujemy. *)
    let reason (s,l)  = ([],l)
  end;;

module Sudoku1 = Sudoku_BT (NoReasoning);;
```

Nie jest to jednak zbyt efektywny algorytm, choć wystarczający do rozwiązywania prostych sudoku dla $n = 3$.

```
module Reasoning =
  struct
    open Sudoku_Framework

    (* Konfiguracja, to plansza i lista pustych pól. *)
    type config =  sudoku * (int * int) list

    (* Sprawdza czy dana wartość występuje na liście. *)
```

```
    let present x l = List.filter (fun y -> x = y) l <> []

    (* Jeżeli w wierszu jest tylko jedno miejsce, w którym może  *)
    (* występować jakaś cyfra, to jest tam ona wstawiana.        *)
    let reason_row (s, l) =
      let changed = ref true
      and puste = ref l
      and wstawione = ref []
      and n = size s
      in
        while !changed do
          changed := false;
          (* Kolejne wiersze. *)
          for i = 0 to n * n - 1 do
            (* Wartości, które mogą pojawić się na poszczególnych miejscach.
*)
            let v =
              Array.init (n*n)
                (fun j -> if s.(i).(j) = 0 then valid_values s i j else
[s.(i).(j)])
            in
              (* Przejrzyj kolejne cyfry. *)
              for k = 1 to n * n do
                (* Sprawdź, na ilu pozycjach może występować cyfra. *)
                let jj = List.filter (fun j -> present k v.(j)) (indeksy s)
                in
                  (* Jeśli pozycja cyfry jest wyznaczona *)
                  (* jednoznacznie, wypełnij ją.         *)
                  match jj with
                    [j] -> if s.(i).(j) = 0 then begin
                      changed := true;
                      s.(i).(j) <- k;
                      wstawione := (i,j)::!wstawione
                    end |
                    _ -> ()
              done
          done;
          (* Korekta listy pustych pól. *)
          puste := List.filter (fun (i,j) -> s.(i).(j)=0) (!puste);
        done;
        (!wstawione, !puste)

    (* Jeżeli w kolumnie jest tylko jedno miejsce, w którym może  *)
    (* występować jakaś cyfra, to jest tam ona wstawiana.         *)
    let reason_col (s, l) =
      let changed = ref true
      and puste = ref l
```

```
      and wstawione = ref []
      and n = size s
      in
        while !changed do
          ⋮
        done;
        (!wstawione, !puste)

    (* Jeżeli w małym kwadracie jest tylko jedno miejsce, w którym         *)
    (* może występować jakaś cyfra, to jest tam ona wstawiana.             *)
    let reason (s,l) =
      ⋮

  end;;

module Sudoku2 = Sudoku_BT (Reasoning);;
```

16.4 Obcinanie gałęzi

Zasadnicze znaczenie dla złożoności back-trackingu ma obcinanie gałęzi nie prowadzących do interesujących nas wyników oraz zmniejszenie liczby rozgałęzień rekurencyjnych. Backtracking ma zwykle koszt wykładniczy. Zmniejszenie liczby rozgałęzień rekurencyjnych odpowiada zmniejszeniu podstawy potęgi wyrażającej koszt czasowy.

Możemy starać się obcinać gałęzie stosując proste kryteria negatywne. Np. proste warunki, z których wynika, że skonstruowanej części rozwiązania nie da się uzupełnić do poprawnego rozwiązania. Jeżeli szukamy rozwiązania pod jakimś względem optymalnego, to możemy starać się oszacować od dołu koszt rozwiązań powstałych z rozwinięcia częściowego rozwiązania. Jeżeli koszt ten jest na pewno większy od kosztu znalezionego już rozwiązania, to nie warto przeglądać takich gałęzi.

Warto również zwrócić uwagę na kolejność rozpatrywania kolejnych wariantów. Należy najpierw wybierać takie, które mają szansę dać lepsze oszacowania, lub lepiej rokują na rozwiązanie. Wówczas mamy szansę wcześniej znaleźć rozwiązanie, lub obciąć więcej gałęzi.

[[Przykład na obcinanie na podstawie juz uzyskanych wyników]]

[[Rozwiązanie początkowe i końcowe są dane – szukamy konfiguracji w środku.]]

Ćwiczenia

- Problem obejścia szachownicy skoczkiem.
- Ustawić minimalną liczbę hetmanów szachujących wszystkie pola na szachownicy,
- Napisz procedurę `pierwsze`, która dla danej liczby całkowitej n ($n > 1$) wyznacza rozbicie n na sumę minimalnej liczby składników będących liczbami pierwszymi. (Możesz założyć prawdziwość hipotezy Goldbacha.)
- Dany jest (multi-)zbiór kostek domina. Należy wyznaczyć maksymalny łańcuch, jaki można ułożyć z danych kostek, oczywiście zgodnie z zasadami domina. Rozwiązanie powinno mieć postać procedury `domino` : $(\alpha \times \alpha)$ `list` $\rightarrow (\alpha \times \alpha)$ `list`. Klocki można obracać.
- [III OI] Mokra robota. Dany jest zestaw naczyń szklanych o określonych objętościach. Dla każdego z nich wiemy ile chcemy, aby było do niego nalane wody. Początkowo wszystkie naczynia są napełnione wodą. Dozwolone operacje to:
  - przelanie całej zawartości jednego naczynia do drugiego, pod warunkiem, że się zmieści,
  - dolanie wody do pełna z jednego naczynia do drugiego,
  - wylanie wody z naczynia do zlewu.

  Szukamy takiej sekwencji ruchów, po wykonaniu której we wszystkich naczyniach jest tyle wody, ile chcieliśmy. (Nie koniecznie back-tracking, ale przeszukiwanie przestrzeni stanów.)
- [XIV OI] Atrakcje turystyczne (uproszczone). Mamy n miejscowości, ponumerowanych od 1 do n. Odległości między miejscowościami są dane w postaci (symetrycznej) tablicy liczb całkowitych, o wymiarach $n \times n$.

  Chcemy zwiedzić wszystkie te miejscowości, jednak nie w dowolnej kolejności. Mamy daną listę ograniczeń postaci $[(i_1, j_1); \ldots; (i_k, j_k)]$. Ograniczenie (i, j) oznacza, że miasto i musi być zwiedzone przed miastem j. Możesz założyć, że ograniczenia da się spełnić.

  Na początku wyruszamy z miasta 1, a kończymy podróż w mieście n. Wyznacz minimalną drogę jaką trzeba przejechać, żeby zwiedzić wszystkie miasta.

Wykład 17. Technika spamiętywania

17.1 Mapy

Koncepcja mapy/słownika/funkcji częściowej.

```
module type MAP =
  sig
    (* Mapa z wartości typu 'a w 'b. *)
    type ('a,'b) map

    (* Wyjątek podnoszony, gdy badamy wartość spoza dziedziny. *)
    exception Undefined

    (* Pusta mapa. *)
    val empty : ('a,'b) map

    (* Predykat charakterystyczny dziedziny mapy. *)
    val dom : ('a,'b) map -> 'a -> bool

    (* Zastosowanie mapy. *)
    val apply : ('a,'b) map -> 'a -> 'b

    (* Dodanie wartości do mapy. *)
    val update : ('a,'b) map -> 'a -> 'b -> ('a,'b) map
  end;;
```

Uwaga: Ocaml zawiera moduł `Map`. Implementuje on nie tylko funkcjonalność przedstawioną tutaj, ale również wiele innych procedur. Choć nazwy niektórych z nich są inne niż tutaj.

17.1.1 Prosta implementacja proceduralna

Mapa to procedura, która punktom z dziedziny przyporządkowuje wartości, a dla pozostałych punktów podnosi wyjątki.

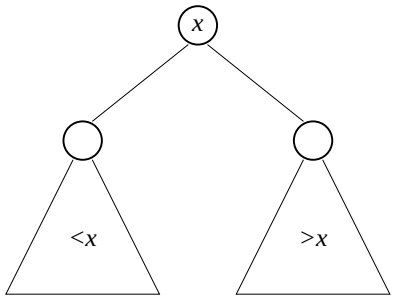
```
module ProcMap : MAP =
  struct
    (* Mapa z wartości typu 'a w 'b. *)
    type ('a, 'b) map = 'a -> 'b

    (* Wyjątek podnoszony, gdy badamy wartość spoza dziedziny. *)
    exception Undefined

    (* Pusta mapa *)
    let empty = function _ -> raise Undefined
    (* Predykat charakterystyczny dziedziny mapy. *)
    let dom m x =
```

```
    try
      let _ = m x
      in true
    with Undefined -> false

  (* Zbadanie wartości w punkcie *)
  let apply m x = m x

  (* Dodanie wartości do mapy. *)
  let update f a v =
    function x ->
      if a = x then v else f x
end;;
```

Jest to oczywiście implementacja nieefektywna.

17.1.2 Drzewa BST

Idea drzew BST. Do zaimplementowania drzew BST potrzebny jest porządek liniowy określony na wartościach, które mogą pojawiać się jako klucze. Ocaml na każdym typie określa porządek liniowy.



Struktura danych. Węzeł ma cztery argumenty: klucz, wartość, lewe i prawe poddrzewo.

```
module BstMap : MAP =
  struct
    (* Mapa z wartości typu 'a w 'b. *)
    type ('a, 'b) map =
      Empty |
      Node of ('a * 'b * ('a, 'b) map * ('a, 'b) map)

    (* Wyjątek podnoszony, gdy badamy wartość spoza dziedziny. *)
    exception Undefined

    (* Pusta mapa to puste drzewo *)
    let empty = Empty

    (* Znajdź poddrzewo o zadanym kluczu              *)
    (* (jeśli nie ma, to wynikiem jest puste drzewo). *)
```

```
  let rec find tree key =
    match tree with
      Empty -> Empty |
      Node (k, _, l, r) ->
        if key = k then tree
        else if key < k then find l key
        else find r key

  (* Zastosowanie mapy. *)
  let apply m k =
    match find m k with
      Empty -> raise Undefined |
      Node (_, v, _, _) -> v

  (* Sprawdzenie, czy punkt należy do dziedziny *)
  let dom m x =
    try
      let _ = apply m x
      in true
    with
      Undefined -> false

  (* Zmiana wartości mapy w punkcie *)
  let rec update tree key value =
    match tree with
      Empty -> Node (key, value, Empty, Empty) |
      Node (k, v, l, r) ->
        if key = k then
          Node (key, value, l, r)
        else if key < k then
          Node (k, v, update l key value, r)
        else
          Node (k, v, l, update r key value)
end;;
```

Pesymistyczna złożoność operacji na drzewie BST jest proporcjonalna do wysokości drzewa, czyli w najgorszym przypadku jest rzędu $O(n)$. Jeśli jednak wstawiane wartości są losowe, to średnia wysokość drzewa jest dużo mniejszego rzędu: $O(\log n)$.

Są też zrównoważone wersje drzew BST (np. AVL), które zawsze gwarantują, że wysokość drzewa jest rzędu $O(\log n)$.

17.2 Technika spamiętywania

Mając do dyspozycji mapy możemy zapisać rozwiązanie w sposób rekurencyjny, unikając jednak wielokrotnych wywołań rekurencyjnych dla tych samych wartości argumentów. Korzystamy w tym celu z techniki *spamiętywania*. Technika ta polega na pamiętaniu wszystkich

obliczonych uprzednio rozwiązań podproblemów. Jest to skrzyżowanie rozwiązania rekurencyjnego i programowania dynamicznego, o którym powiemy dokładniej w kolejnym wykładzie.

Zobaczmy, jak wygląda zastosowanie opisanej techniki do obliczania liczb Fibonacciego:

```
let tab = ref empty;;
let rec fib n =
  if dom !tab n then
    apply !tab n
  else
    let wynik = if n < 2 then n else fib (n-1) + fib (n-2)
    in begin
      tab := update !tab n wynik;
      wynik
    end;;
```

Schemat spamiętywania jest dosyć uniwersalny: W implementacji rekurencyjnej, ilekroć chcemy wywołać rekurencyjnie procedurę rozwiązującą podproblem, sprawdzamy, czy już takie wywołanie nie miało miejsca, a jeżeli tak, to pobieramy obliczony wcześniej wynik. Możemy zaimplementować coś w rodzaju „otoczki", która filtruje wywołania procedur. Mechanizm ten możemy wyłuskać i przedstawić w postaci procedury wyższego rzędu. Parametrami tej procedury są: tablica użyta do spamiętywania, spamiętywana funkcja i jej argument.

```
let memoize tab f x =
  if dom !tab x then
    apply !tab x
  else
    let wynik = f x
    in begin
      tab := update !tab x wynik;
      wynik
    end;;
```

Ponownie, procedurę obliczająca liczby Fibonacciego możemy zapisać następująco:

```
let fib =
  let tab = ref empty
  in
    let rec f n =
      memoize tab (function n ->
        if n < 2 then n else f (n-1) + f (n-2)) n
    in f;;
```

17.3 Problem NWP 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 różny od x_i, to $nwp(i,j) = nwp(i-1, j)$.
- Jeśli ostatni wyraz nwp jest różny od y_j, to $nwp(i,j) = nwp(i, j-1)$.
- Funkcja nwp jest monotoniczna względem obydwu współrzędnych.
- Stąd, jeśli $x_i \neq y_j$, to $nwp(i,j) = \max(nwp(i-1, j), nwp(i, j-1))$.
- Warunki brzegowe — zero.

- Algorytm — tabela rozwiązań dla podproblemów i jej wypełnianie.

		A	**B**	**C**	**B**	**D**	**A**	**B**
	0	0	0	0	0	0	0	0
B	0	0	**1**	1	**1**	1	1	**1**
D	0	0	1	1	1	**2**	2	2
C	0	0	1	**2**	2	2	2	2
A	0	**1**	1	2	2	2	**3**	3
B	0	1	**2**	2	**3**	3	3	**4**
A	0	**1**	2	2	3	3	**4**	4

Wszystkie najdłuższe wspólne podciągi to:

- `BCBA`,
- `BCAB`,
- `BDAB`,

przy czym ostatni z nich występuje w pierwszym ciągu na dwa sposoby.

17.4 Najdłuższy wspólny podciąg — implementacja ze spamiętywaniem

W poniższym programie mapa przyporządkowuje parom liczb (i, j) pary (najdłuższy wspólny podciąg, jego długość). Wynikiem procedury `pom` jest para: (odwrócony) najdłuższy wspólny podciąg i jego długość. Jako efekt uboczny obliczeń powstaje również mapa zawierająca wszystkie obliczone wyniki podproblemów. Wynikiem całej procedury jest najdłuższy wspólny podciąg dwóch ciągów.

```
let nwp ciag_x ciag_y =
  let x = Array.of_list ciag_x
  and y = Array.of_list ciag_y
  and tab = ref empty
  in
    let rec pom (i, j) =
      memoize tab (fun (i, j) ->
        if i = 0 or j = 0 then ([], 0)
        else if x.(i-1) = y.(j-1) then
          let (c, l) = pom (i-1, j-1)
```

```
          in (x.(i-1)::c, l+1)
        else
          let (c1, l1) = pom (i-1, j)
          and (c2, l2) = pom (i, j-1)
          in
            if l1 > l2 then (c1, l1) else (c2, l2)
      ) (i, j)
    in
      let (w, _) = pom (length ciag_x, length ciag_y)
      in rev w;;

nwp ["A"; "B"; "C"; "B"; "D"; "A"; "B"]
    ["B"; "D"; "C"; "A"; "B"; "A"];;
- : string list = ["B"; "D"; "A"; "B"]
```

Laboratorium

Rozszerz implementację drzew BST o następujące operacje:

- `segment_min` s a b oblicza minimalny element x należący do s taki, że $a \leq x \leq b$ (jeśli brak takiego elementu, należy podnieść wyjątek `Undefined`),
- `segment_max` s a b oblicza maksymalny element x należący do s taki, że $a \leq x \leq b$ (jeśli brak takiego elementu, należy podnieść wyjątek `Undefined`),
- `segment_size` s s b oblicza liczbę takich elementów x należących do s, że $a \leq x \leq b$,
- `find_nth` s r znajduje w s element o randze r, tzn. r-ty co do wielkości.

Wszystkie te operacje powinny działać w czasie proporcjonalnym do wysokości drzewa. Czas 2 tygodnie, 3 punkty.

Ćwiczenia

- Napisz procedurę, która przekształca dane drzewo binarne w wyważone drzewo binarne, zachowując kolejność elementów w porządku infiksowym.
- Jaka jest minimalna liczba monet potrzebna do wydania n zł reszty, przyjmując, że w obrocie są dostępne monety o zadanych (w postaci listy) nominałach?
- Na ile sposobów można wydać n zł reszty przyjmując, że w obrocie są dostępne monety o zadanych (w postaci listy) nominałach.
- [XII OI, Banknoty] W bankomacie znajdują się banknoty o nominałach $b_1, b_2, \ldots, b_n$, przy czym banknotów o nominale b_i jest w bankomacie c_i. Jak wypłacić zadaną kwotę używając jak najmniejszej liczby banknotów?
- Napisz procedurę, która dla danego ciągu liczb wyznaczy jego najdłuższy rosnący podciąg.
- Napisz procedurę `pierwsze`, która dla danej liczby całkowitej n ($n > 1$) wyznacza rozbicie n na sumę minimalnej liczby składników będących liczbami pierwszymi. (Tym razem nie zakładaj prawdziwości hipotezy Goldbacha.)
- Dana jest liniowa plansza z polami ponumerowanymi od 1 do n, obdarzonymi rzeczywistymi wagami $a_1, a_2, \ldots, a_n$. Na polu 1 stoi pionek, który może poruszać się po planszy skokami wprzód lub w tył na odległość nieprzekraczającą k, aż zatrzyma się definitywnie na polu n (pionek może odwiedzić pola 1 i n więcej niż raz). Każde pole, na którym stanie pionek jest zbite.

  Napisać funkcję `chocki`, która dla danego k oraz listy $(a_1 a_2 \ldots a_n)$ zwróci maksymalną możliwą sumę wag pól niezbitych.

Wykład 18. Programowanie dynamiczne

18.1 Implementacja imperatywna, z wykorzystaniem tablic

W przypadku problemu najdłuższego wspólnego podciągu, możemy spamiętywanie zaimplementować wprost, gdyż najistotniejszym wynikiem jest tablica, która powstaje w trakcie obliczeń.

```
let nwp ciag_x ciag_y =
  if (ciag_x = []) || (ciag_y = []) then [] else
  let n = length ciag_x
  and m = length ciag_y
  and x = Array.of_list ((hd ciag_x)::ciag_x)
  and y = Array.of_list ((hd ciag_y)::ciag_y)
  in
    (* a.(i).(j) = length (nwp [x_1;..;x_i] [y_1;..;y_j]) *)
    let a = Array.make_matrix (n+1) (m+1) 0
    in
      (* Rekonstrukcja nwp na podstawie tablicy długości. *)
      let rec rekonstrukcja acc i j =
        if i = 0 or j = 0 then acc
        else if x.(i) = y.(j) then
          rekonstrukcja (x.(i)::acc) (i-1) (j-1)
        else if a.(i).(j) = a.(i-1).(j) then
          rekonstrukcja acc (i-1) j
        else
          rekonstrukcja acc i (j-1)
      in
        begin
          (* Wypełnienie tablicy a *)
          for i = 1 to n do
            for j = 1 to m do
              if x.(i) = y.(j) then
                (* Odpowiadające sobie elementy ciągów są równe. *)
                a.(i).(j) <- a.(i-1).(j-1) + 1
              else
                (* Wybór wariantu realizującego dłuższy podciąg. *)
                a.(i).(j) <- max a.(i-1).(j) a.(i).(j-1)
            done
          done;
          (* Rekonstrukcja wyniku. *)
          rekonstrukcja [] n m
        end;;

nwp ["A"; "B"; "C"; "B"; "D"; "A"; "B"] ["B"; "D"; "C"; "A"; "B"; "A"];;
```

18.2 Implementacja NWP oparta o mapy

- Główna procedura:

```
let nwp ciag_x ciag_y =
  let
    a = zbuduj_tablice1 ciag_x ciag_y
  in
    odtworz_podciag1 a ciag_x ciag_y;;
```

- Budowa tablicy:

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

let zbuduj_tablice ciag_x ciag_y =

  (*
     Dodaje kolejny wiersz tablicy,
     y= rozpatrywany element ciągu y, j = nr wiersza. *)
  let dodaj_wiersz tablica y j =
    let (t, _) =
      fold_left
        (fun (tab, i) x ->
          let u = apply tab (j, i-1)
          and v = apply tab (j-1, i-1)
          and w = apply tab (j-1, i)
          in
            (update tab (j, i) (dlugosc u v w x y), i+1))
        (update tablica (j, 0) 0, 1)
        ciag_x
    in t
  in
    (* Pierwszy wiersz tablicy - same zera *)
    let pierwszy n =
```

```
  let (a, _) =
    fold_left
      (fun (a, i) _ -> (update a (0, i) 0, i+1))
      (update empty (0, 0) 0, 1)
      ciag_x
  in a

(* Budowanie tablicy,
   yl= pozostale elementy ciągu y, j = nr wiersza. *)
and buduj a yl =
  let (w, _) =
    fold_left (fun (a, j) y -> (dodaj_wiersz a y j, j+1)) (a, 1) yl
  in w
in
  buduj (pierwszy (length ciag_x)) ciag_y;;
```

- Rekonstrukcja podciągu:

```
(* Rekonstrukcja ciągu na podstawie tablicy *)
let odtworz_podciag tablica ciag_x ciag_y =
  let rec odtworz akumulator tab ciag_x ciag_y i j =
    match ciag_x with
      [] -> akumulator |
      (x::tx) ->
        match ciag_y with
          [] -> akumulator |
          (y::ty) ->
            if x = y then
              odtworz (x::akumulator) tab tx ty (i-1) (j-1)
            else
              let u = apply tab (j, i-1)
              and w = apply tab (j-1, i)
              in
                if u > w then
                  odtworz akumulator tab tx ciag_y (i-1) j
                else
```

```
                odtworz akumulator tab ciag_x ty i (j-1)
in
  odtworz [] tablica
          (rev ciag_x) (rev ciag_y)
          (length ciag_x) (length ciag_y);;
```

18.3 Efektywna funkcyjna implementacja NWP, efektywna, acz skomplikowana

- Struktura tablicy

```
type array = { value : int; up : array; upleft : array; left : array };;
let rec zero = { value = 0; up = zero; upleft = zero; left = zero };;
```

- Wypełnianie tablicy

```
(* A[i, j] = długość najdłuższego wspólnego podciągu      *)
(* (x_1, ..., x_i) i (y_1, ..., y_j).                     *)
(* Tablica jest reprezentowana w postaci odwróconej listy *)
(* odwróconych list i uzupełniona strażnikami równymi 0.  *)
let zbuduj_tablice ciag_x ciag_y =
  (* Dodaje kolejny wiersz tablicy,
     y= rozpatrywany element ciągu y.
  *)
  let dodaj_wiersz tab y  =
    let rec dodawaj tab_up lista =
      match lista with
        []     -> zero |
        (x::t) ->
          let tab_upleft = tab_up.left
          in
            let tab_left = dodawaj tab_upleft t
            in
              let v = dlugosc tab_left.value
                              tab_upleft.value
                              tab_up.value
                              x y
              in
```

```
                { value  = v;
                  left   = tab_left;
                  upleft = tab_upleft;
                  up     = tab_up
                }
    in
      dodawaj tab (rev ciag_x)
  in
    fold_left dodaj_wiersz zero ciag_y;;
```

- Odtworzenie ciągu na podstawie tablicy długości

```
let odtworz_podciag tablica ciag_x ciag_y =
  let rec odtworz akumulator tab ciag_x ciag_y =
    match ciag_x with
      [] -> akumulator |
      (x::tx) ->
        match ciag_y with
          [] -> akumulator |
          (y::ty) ->
            if x = y then
              odtworz (x::akumulator) tab.upleft tx ty
            else
              let lv = tab.left.value
              and uv = tab.up.value
              in
                if lv > uv then
                  odtworz akumulator tab.left tx ciag_y
                else
                  odtworz akumulator tab.up ciag_x ty
  in
    odtworz [] tablica (rev ciag_x) (rev ciag_y);;
```

Ćwiczenia

- [XII OI] Ulice miasta tworzą szachownicę — prowadzą z północy na południe, lub ze wschodu na zachód, a każda ulica prowadzi na przestrzał przez całe miasto — każda ulica biegnąca z północy na południe krzyżuje się z każdą ulicą biegnącą ze wschodu na zachód i vice versa. Ulice prowadzące z północy na południe są ponumerowane od 1 do n, w kolejności z zachodu na wschód. Ulice prowadzące ze wschód na zachód są ponumerowane od 1 do m, w kolejności z południa na północ. Każde skrzyżowanie i-tej ulicy biegnącej z północy na południe i j-tej ulicy biegnącej ze wschodu na zachód oznaczamy parą liczb (i, j) (dla $1 \leq i \leq n$, $1 \leq j \leq m$).

  Po ulicach miasta kursuje autobus. Zaczyna on trasę przy skrzyżowaniu $(1, 1)$, a kończy przy skrzyżowaniu (n, m). Ponadto autobus może jechać ulicami tylko w kierunku wschodnim i/lub północnym.

  Przy niektórych skrzyżowaniach znajdują się pasażerowie oczekujący na autobus. Rozmieszczenie pasażerów jest dane w postaci listy l trójek postaci (i, j, k). Każda taka trójka oznacza, że przy skrzyżowaniu (i, j) czeka k pasażerów.

  Napisz procedurę `autobus`, która na podstawie liczb n, m oraz listy l obliczy maksymalną liczbę pasażerów, których może zabrać autobus. Zakładamy, że w autobusie zmieści się dowolna liczba pasażerów.

- [CLR, ćw. 16-4] Firma planuje zorganizować przyjęcie. Hierarchia stanowisk w firmie ma strukturę drzewa (ogólnego). Każdy pracownik charakteryzuje 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?

```
type tree = Node of (int * tree list);;
let rec impreza (Node (wsp, sons)) =
  let lp = map impreza sons
  in
    fold_left (fun (z, bez) (x, y) -> (z + y, bez + max x y))
      (wsp, 0) lp;;
```

- Dana jest deklaracja typu:

```
type 'a drzewo =
  Node of 'a * 'a drzewo * 'a drzewo |
  Leaf
```

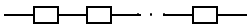
  Napisz procedurę `środek :'a drzewo -> 'a drzewo`, która znajduje w zadanym drzewie taki węzeł (`Node`), dla którego maksymalna spośród jego odległości od liści jest jak najmniejsza. Wynikiem tej procedury powinno być poddrzewo zakorzenione w wyznaczonym węźle.

- [X OI] Zadanie „Płytki drukowane”. Firma Bajtel rozpoczyna produkcję elektronicznych układów szeregowo-równoległych. Każdy taki układ składa się z części elektronicznych, połączeń między nimi oraz dwóch połączeń doprowadzających prąd. Układ szeregowo-równoległy może składać się z:

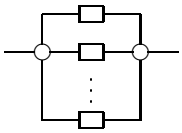
  – pojedynczej części,

  – kilku mniejszych układów szeregowo-równoległych połączonych szeregowo,

  – dwóch części rozgałęziających łączących równolegle kilka mniejszych układów szeregowo-równoległych.

  Układy są montowane na dwustronnych płytkach drukowanych. Problem polega na ustaleniu, które połączenia powinny znaleźć się na górnej, a które na dolnej stronie płytki. Ze względów technicznych, jak najwięcej połączeń powinno znaleźć się na dolnej stronie płytki, jednak do każdej części musi dochodzić przynajmniej jedno połączenie znajdujące się na górnej stronie płytki.

  Struktura układu szeregowo-równoległego jest opisana za pomocą wartości następującego typu:

```
type układ =
  Element |
  Szeregowo of układ list |
  Równolegle of układ list;;
```

  Napisz procedurę `płytki` : `układ` → `int`, która dla danego układu wyznaczy minimalną liczbę połączeń, które muszą być umieszczone na górnej stronie płytki.

- [BOI'2003, przeformułowane] Tomcio chce zrobić model drzewa (acyklicznego nieskierowanego grafu spójnego) z nitki i szklanych koralików i to taki, w którym koraliki tego samego koloru nie sąsiadują ze sobą. Nitkę już ma, ale koraliki musi kupić za swoje kieszonkowe. Ceny koralików różnych kolorów są różne, dla każdego dodatniego całkowitego n w sprzedaży są koraliki jednego koloru w cenie n gr za koralik. Tomcio zastanawia się z jakich koralików zrobić model drzewa, tak aby wydać jak najmniej.

  Opis drzewa ma postać struktury listowej, w której każdy węzeł to lista jego synów. Napisz procedurę `koraliki`, która na podstawie opisu drzewa obliczy minimalny koszt koralików potrzebnych do wykonania jego modelu.

  Uwaga: Dwa kolory wystarczą do wykonania modelu, ale nie dają minimalnego kosztu.

- [IX OI] Zadanie „Waga”. Dany jest zestaw odważników. Czy można na szalkach wagi położyć część odważników tak, żeby waga nadal była zrównoważona? Jeśli tak, to jaki najcięższy odważnik można przy tym użyć?

- [IX OI] Zadanie „Nawiasy”. Dane jest wyrażenie postaci $x_1 \pm x_2 \pm \cdots \pm x_n$. Na ile różnych sposobów można uzyskać wyrażenie równoważne danemu wstawiając nawiasy do wyrażenia $x_1 - x_2 - \cdots - x_n$? Uwaga: wstawiamy $n - 1$ par nawiasów w pełni określając kolejność wykonywania odejmowań.

- [XI OI] Zadanie „Sznurki”. Interesują nas modele drzew (spójnych grafów acyklicznych) wykonane ze sznurka. Każde drzewo składa się z wierzchołków i pewnej liczby krawędzi łączących różne wierzchołki. Ten sam wierzchołek może być połączony z wieloma innymi wierzchołkami. Wierzchołki grafów mają być modelowane przez węzły zawiązane na kawałkach sznurka, a krawędzie przez odcinki sznurka pomiędzy węzłami. Każdy węzeł może być wynikiem zasuplenia kawałka sznurka lub związania w tym samym miejscu wielu kawałków. Każda krawędź ma długość 1. Długość sznurka użytego do zrobienia węzłów jest pomijalna.

  Chcemy zoptymalizować liczbę i długość kawałków sznurka użytych do wykonania modelu drzewa:

  – w pierwszym rzędzie należy zminimalizować liczbę potrzebnych kawałków sznurka,
  – zakładając, że liczba kawałków sznurka jest minimalna, należy zminimalizować długość najdłuższego kawałka sznurka.

Wykład 19. Kody Huffmana i algorytmy zachłanne

19.1 Rodzaje kodów

- Kody stałej długości (np. ASCII).
- Kody zmiennej długości (np. Morse'a).
- Problem separatora — kody prefiksowe.

Przykład: [SICP] Kod zmiennej długości może być efektywniejszy niż kod stałej długości.

A 000	C 010	E 100	G 110
B 001	D 011	F 101	H 111

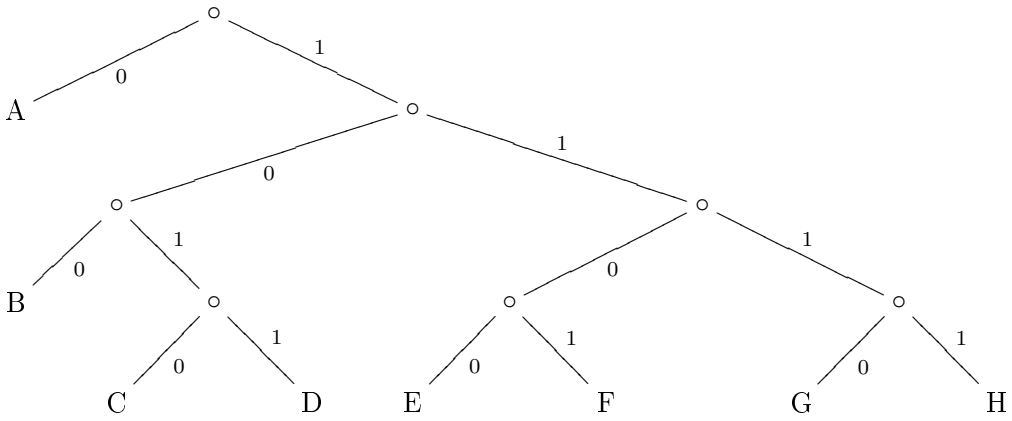
Przy takim kodowaniu, wiadomość:
BACADAEAFABBAAAGAH
jest zakodowana jako ciąg 54 bitów:
001000010000011000100000101000001001000000000110000111

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 ok. 20% pamięci.

- Problem zdefiniowania optymalnego kodu prefiksowego przy znanych (względnych) częstościach występowania znaków.
- Reprezentacja optymalnego kodowania w postaci regularnego drzewa binarnego.

Przykład: Kod z poprzedniego przykładu możemy przedstawić jako drzewo. To że kod jest prefiksowy odpowiada temu, że kodowane znaki odpowiadają liściom.



19.2 Algorytm Huffmana

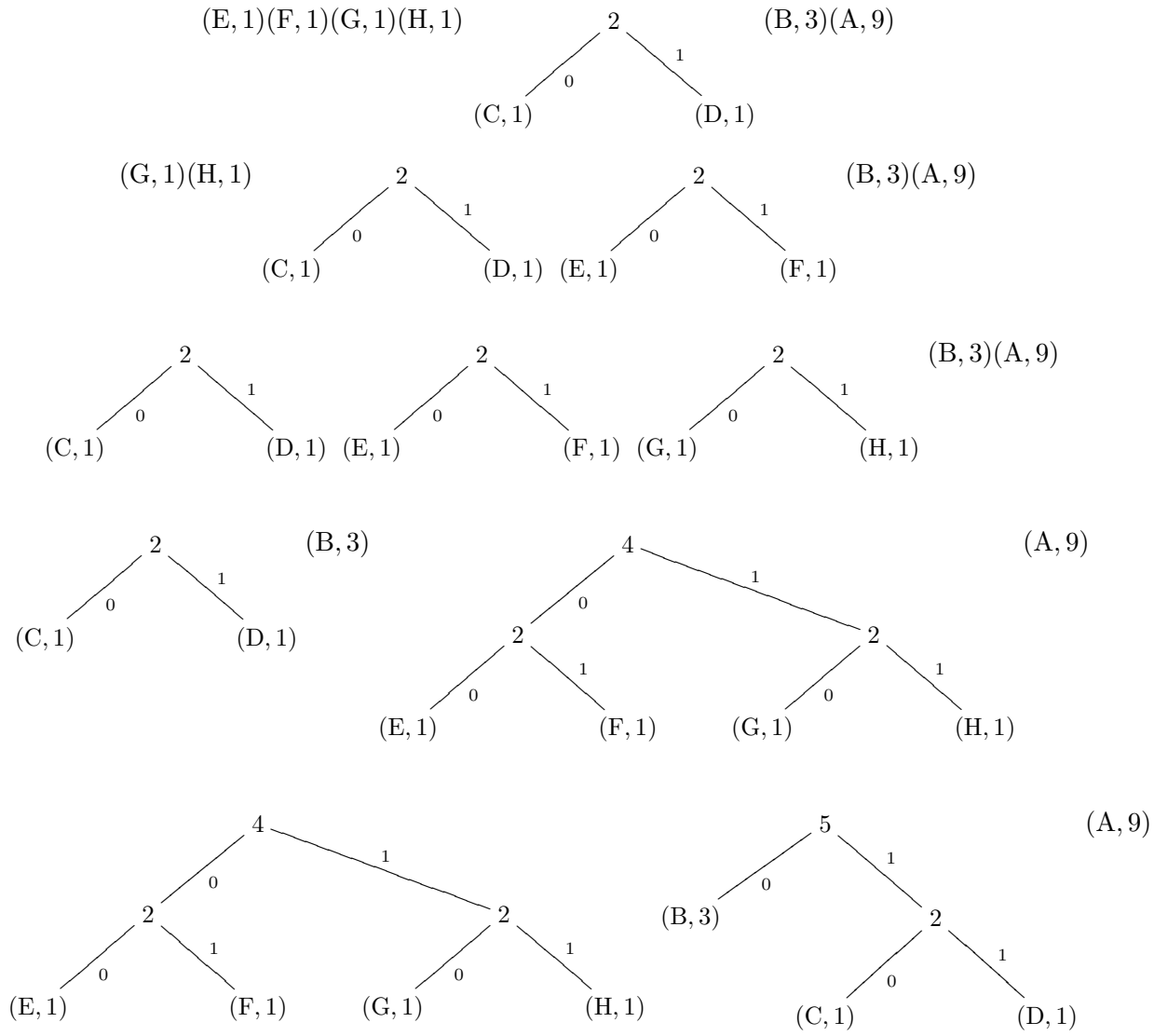
Algorytm Huffmana polega na budowaniu drzewa kodowania od liści do korzenia. Na początku znamy zbiór liści — jest to zbiór wszystkich kodowanych znaków.

W każdym kolejnym kroku wybieramy dwa elementy o najmniejszych częstościach występowania i sklejamy je, tak że ich kody będą się różnić ostatnim bitem. Tak powstały nowy element zastępuje sklejone elementy, a jego częstość występowania jest równa sumie częstości występowania sklejonych elementów.

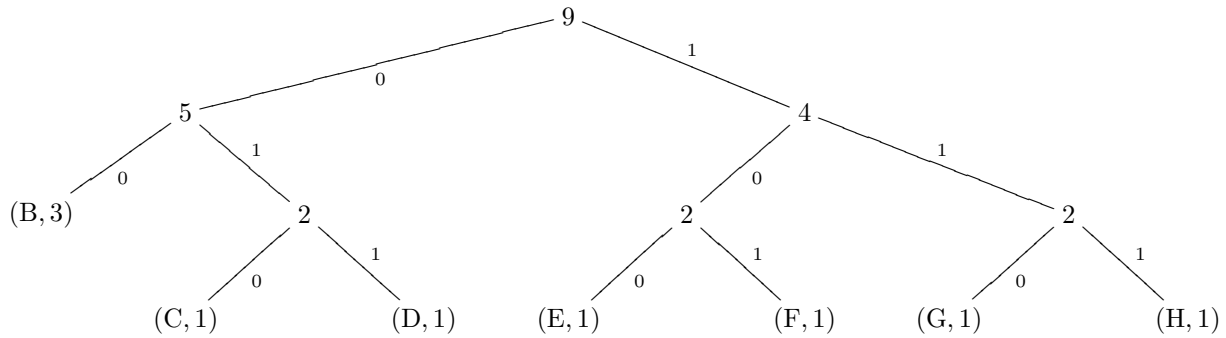
Algorytm kończymy, gdy zostanie nam już tylko jeden element — drzewo kodowania.

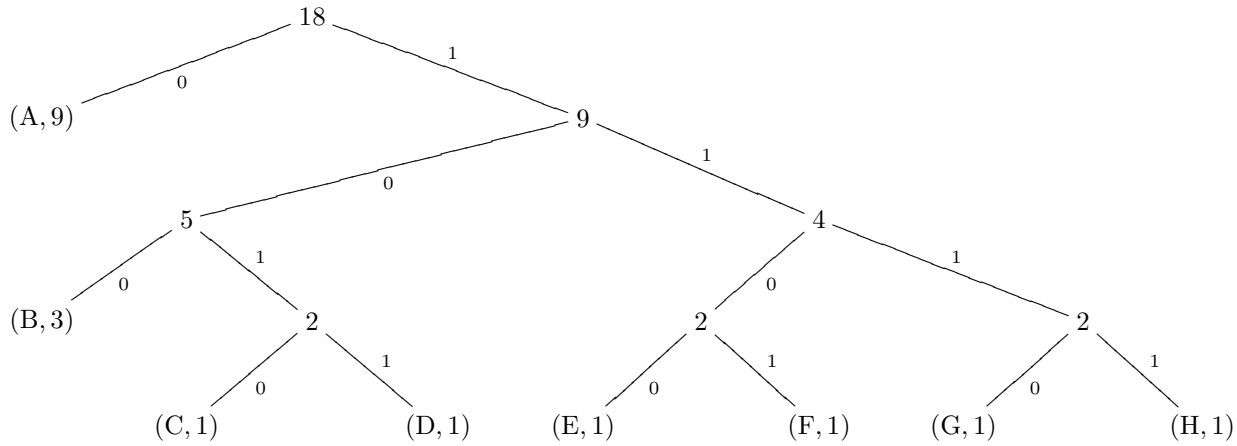
Przykład: Prześledźmy działanie algorytmu dla przykładowych danych.

$$(C,1)(D,1)(E,1)(F,1)(G,1)(H,1)(B,3)(A,9)$$



(A, 9)





Dowód poprawności:

Lemat [CLR] Niech C będzie alfabetem, a każdy znak $c \in C$ występuje $f(c)$ razy. Niech x i y będą parą znaków z C o najmniejszej liczbie wystąpień. Istnieje wtedy optymalny kod prefiksowy dla C, w którym kody dla x i y mają tę samą długość i różnią się tylko ostatnim bitem.

Dowód: Rozważmy pewne optymalne kodowanie. Niech b i c będą dwoma sąsiednimi liśćmi o maksymalnej głębokości $h_b = h_c$. Bez straty ogólności możemy założyć, że $f(b) \leq f(c)$ oraz $f(x) \leq f(y)$. Ponadto mamy $f(x) \leq f(b)$ i $f(y) \leq f(c)$, a głębokości x i y nie przekraczają głębokości b i c, $h_x, h_y \leq h_b = h_c$. Zamieniamy b z x i c z y. Długość kodu zmienia się o:

$$f(x)(h_b - h_x) + f(y)(h_c - h_y) + f(b)(h_x - h_b) + f(c)(h_y - h_c) =$$

$$(f(x) - f(b))(h_b - h_x) + (f(y) - f(c))(h_c - h_y) \leq 0$$

Tak powstałe drzewo daje nie gorsze kodowanie, a więc również optymalne.

Lemat Niech C będzie alfabetem, f będzie funkcją określającą częstość występowania symboli, x i y będą różnymi symbolami o najmniejszych częstościach występowania. Niech z będzie nowym symbolem. Przyjmijmy $f(z) = f(x) + f(y)$. Niech T będzie regularnym drzewem binarnym odpowiadającym optymalnemu kodowaniu dla alfabetu $(C \setminus \{x, y\}) \cup \{z\}$ i f.

Wówczas drzewo T' powstałe przez dodanie do wierzchołka z synów x i y jest optymalnym drzewem kodowania dla C i f

Dowód: Gdyby T' nie było optymalne, to istniałoby lepsze drzewo kodowania, w którym x i y byliby braćmi. Oznaczając ich ojca przez z i usuwając je uzyskalibyśmy drzewo kodowania lepsze od T, co nie jest możliwe.

Z tych dwóch lematów wynika natychmiast poprawność algorytmu.

19.3 Implementacja algorytmu Huffmana

Zgodnie z poprzednim opisem implementujemy algorytm Huffmana:

```
let huffman l =
  let rec process col =
    if huff_size col = 1 then
      huff_first col
    else
      let t1 = huff_first col
      and col1 = huff_remove col
      in
        let t2 = huff_first col1
        and col2 = huff_remove col1
        in
          process (huff_put col2 (merge t1 t2))
  in
    process (make_huff_col l);;
```

Zakładamy tu, że elementy listy `l` są posortowane niemalejąco wg częstości występowania. Korzystamy tu z drzew Huffmana z operacją `merge` oraz kolejek priorytetowych drzew Huffmana, z operacjami: `make_huff`, `huff_size`, `huff_first`, `huff_put` i `huff_remove`.

19.4 Implementacja drzew Huffmana

```
type 'a huff_tree =
  Letter of ('a * float) |
  Node of ('a huff_tree * 'a huff_tree * float);;

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

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

19.5 Implementacja kolekcji drzew Huffmana

```
type 'a huff_col = 'a huff_tree fifo * 'a huff_tree fifo;;

let make_huff_col l =
  (fold_left (fun q x -> put q (Letter x)) empty_queue l, empty_queue);;

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

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

let huff_first ((q1, q2): 'a huff_col) =
  if q1 = empty_queue then first q2 else
  if q2 = empty_queue then first q1 else
    let f1 = first q1
    and f2 = first q2
    in
      if frequency f1 <= frequency f2 then f1 else f2;;

let huff_remove ((q1, q2): 'a huff_col) =
  if q1 = empty_queue then (q1, remove q2) else
  if q2 = empty_queue then (remove q1, q2) else
    let f1 = first q1
    and f2 = first q2
    in
      if frequency f1 <= frequency f2 then
        (remove q1, q2)
      else
        ((q1, remove q2): 'a huff_col);;
```

Zasymuluj działanie algorytmu Huffmana pokazując zawartość kolejek implementujących kolekcję drzew.

19.6 Schemat programowania zachłannego

- szukamy pewnego rozwiązania optymalnego, robimy to za pomocą ciągu wyborów („lokalnych") kolejnych elementów rozwiązania; analogia ze stertą cegieł i zbudowaniem jak największego domu z 1000 cegieł,
- własność wyboru zachłannego — ciąg wyborów optymalnych lokalnie prowadzi nas do rozwiązania optymalnego globalnie,
- jeśli konstruujemy optymalne rozwiązanie przez dokonywanie kolejnych wyborów, to taką własność zwykle dowodzimy indukcyjnie: pokazujemy, że jeśli istnieje rozwiązanie optymalne zgodne z dokonanymi do tej pory wyborami, to istnieje rozwiązanie optymalne zgodne również z kolejnym wyborem (wliczając w to sytuację, gdy dokonujemy pierwszego wyboru),

- jeśli konstruujemy programy zgodnie z zasadą „dziel i zwyciężaj”, to zwykle własności zachłannego wyboru dowodzimy w oparciu o własność optymalnej podstruktury — optymalne rozwiązanie jest funkcją optymalnych rozwiązań podproblemów i łączącego je zachłannego wyboru,

Ćwiczenia

- ciągły problem plecakowy,
- [CLR] p. 17.1, problem wyboru zajęć (historyjka o kinomanie i jak największej liczbie filmów do obejrzenia),
- 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.

  Liczba gwoździ jest taka sama jak liczba seansów filmowych. Oba problemy rozwiązuje ten sam algorytm. Nie widać jednak ogólnej odpowiedniości między tymi zadaniami.
- [CLR] ćw 17.2-4, problem tankowania na stacjach benzynowych,
- [XII OI] Bajtazar postanowił polecieć na Marsa, aby zwiedzić istniejące tam stacje badawcze. Wszystkie stacje na Marsie leżą na okręgu. Bajtazar ląduje w jednej z nich, a następnie porusza się za pomocą specjalnego pojazdu, który jest napędzany odpowiednim paliwem. Litr paliwa starcza na metr jazdy. Zapasy paliwa są jednak niewielkie, różne jego ilości znajdują się w różnych stacjach. Bajtazar może tankować paliwo na stacji, na której w danym momencie się znajduje, nie więcej jednak, niż dostępna tam jego ilość (pojemność baku jest nieograniczona). Musi mu to wystarczyć na dojazd do następnej stacji. Bajtazar musi zdecydować, gdzie powinien wylądować, tak żeby mógł zwiedzić wszystkie stacje. Na koniec Bajtazar musi wrócić do stacji, w której wylądował. W czasie podróży Bajtazar musi poruszać się po okręgu, stale w wybranym jednym z dwóch kierunków.
- Rozplanować zbiór zajęć po salach. Jaka jest minimalna liczba potrzebnych sal?
- Dana jest lista liczb rzeczywistych $[x_1; x_2; \ldots; x_n]$. Napisz procedurę `przekładaniec`, której wynikiem jest taka permutacja $[x_{p_1}; x_{p_2}; \ldots; x_{p_n}]$ danej listy, dla której suma

  $$\sum_{i=1}^{n-1} |x_{p_{i+1}} - x_{p_i}|$$

  jest największa.
- Firma X podjęła się szeregu zobowiązań, z których wykonaniem zalega. Wykonanie każdego z nich zajmuje określoną liczbę dni. Za każdy dzień opóźnienia wykonania każdego z zobowiązań firma płaci określone kary. Napisz procedurę `harmonogram` która obliczy optymalną kolejność wykonywania zaległych zobowiązań. Procedura ta otrzymuje listę trójek postaci:

  (*nazwa czynności*, *liczba dni*, *kara za dzień zwłoki*)

  Jej wynikiem powinna być lista nazw czynności w kolejności, w jakiej należy je wykonywać.

Wykład 20. Algorytmy zachłanne c.d.

20.1 Problem kajakowy

Problem[3]**:** Jak usadzić n osób o podanych wagach w minimalnej liczbie kajaków? Kajaki są dwumiejscowe i mają określoną wyporność. W kajaku może płynąć tylko jedna lub dwie osoby. Zakładamy, że waga żadnej z osób nie przekracza wyporności kajaka, ale dwie osoby mogą już przekroczyć wyporność kajaka. Wagi osób są dane w postaci listy uporządkowanej niemalejąco.

Problem ten możemy rozwiązać zachłannie i to na kilka sposobów. W rozwiązaniach tych będziemy korzystać z dwustronnej kolejki FIFO-LIFO. Dla uproszczenia zakładamy, że dane są posortowane niemalejąco.

1. Najgrubszego kajakarza nazwiemy grubasem. Tych spośród pozostałych kajakarzy, którzy się mieszczą z nim w kajaku nazwiemy chudzielcami. Pozostałych również nazwiemy grubasami.

   Idea algorytmu polega na tym, że największemu grubasowi znajdujemy jak najgrubszego chudzielca, który może płynąć razem z nim. Czyli sadzamy razem największego grubasa i największego chudzielca. Zauważmy przy tym, że im chudszy grubas, tym grubszy może być chudzielec. Tak więc podział na grubasów i chudzielców będzie zmieniał się w czasie — chudsze grubasy będą przechodzić do puli chudzielców, a w razie braku grubasów, najgrubszy chudzielec może zostać uznany za grubasa.

   Początkowo o wszystkich zakładamy, że są grubi i korygujemy podział.

```
let kajaki l wyp =
  let rec dobierz g ch =
    if (is_empty_queue g) && (is_empty_queue ch) then
      (g, ch) else
    if is_empty_queue g then
      (make_queue [last ch], remove_last ch) else
    if queue_size g = 1 then (g, ch) else
    if first g + last g <= wyp then
      dobierz (remove_first g) (put_last ch (first g))
    else (g, ch)
  in
    let rec sadzaj gp chp acc =
      let (g, ch) = dobierz gp chp
      in
        if is_empty_queue g then acc else
        if is_empty_queue ch then
```

[3]Podane sformułowanie problemu pochodzi z zadania *Kajaki* z IV OI. Problem ten został wcześniej sformułowany niezależnie przez autora w równoważnej postaci.

```
          sadzaj (remove_last g) ch ([last g]::acc)
        else
          sadzaj
            (remove_last g)
            (remove_last ch)
            ([last g; last ch]::acc)
    in
      sadzaj (make_queue l) empty_queue [];;

kajaki [1; 2; 2; 3; 3; 3; 4; 6; 6; 8; 8; 8; 9; 9; 10] 10;;
[[3; 3]; [6; 3]; [6; 4]; [8]; [8; 2]; [8; 2]; [9]; [9; 1]; [10]]
```

Uzasadnienie poprawności.

2. Najchudszy kajakarz jest chudzielcem. Grubasy, to ci, którzy nie mieszczą się z nim w kajaku. Pozostali to chudzielce. Jeśli jest tylko jeden chudy, to przyjmujemy, że jest on gruby.

   Idea algorytmu polega, żeby najchudszego chudzielca posadzić z jak najgrubszym kajakarzem, czyli najgrubszym chudzielcem. Początkowo o wszystkich zakładamy, że są chudzi, a następnie przerzucamy ich do puli grubych. W miarę usadzania w kajakach część chudych przechodzi do puli grubych. kajakarze, którzy raz trafią do puli grubych na pewno będą siedzieć samotnie w kajaku. Na koniec grubasy siadają samotnie.

```
let kajaki l wyp =
  let rec dobierz ch g =
    if is_empty_queue ch then (ch, g) else
    if queue_size ch = 1 then
      (empty_queue, put_first g (last ch)) else
    if first ch + last ch > wyp then
      dobierz (remove_last ch) (put_first g (last ch))
    else
      (ch, g)
in
  let rec sadzaj chp gp acc =
    let (ch, g) = dobierz chp gp
    in
      if (is_empty_queue ch) && (is_empty_queue g) then acc else
      if is_empty_queue ch then
        sadzaj ch (remove_first g) ([first g]::acc)
```

```
      else
        sadzaj (remove_first (remove_last ch)) g
               ([first ch; last ch]::acc)
  in
    sadzaj (make_queue l) empty_queue [];;

kajaki [1; 2; 2; 3; 3; 3; 4; 6; 6; 8; 8; 8; 9; 9; 10] 10;;
[[10]; [9]; [8]; [3; 4]; [3; 6]; [3; 6]; [2; 8]; [2; 8]; [1; 9]]
```

Uzasadnienie poprawności.

3. Najgrubszego kajakarza sadzamy z najchudszym, o ile się zmieszczą. Jeśli nie, to najgrubszy kajakarz płynie sam.

```
let kajaki l wyp =
  let rec sadzaj q acc =
    if is_empty_queue q then acc else
    if queue_size q = 1 then [first q]::acc else
    if first q + last q <= wyp then
      sadzaj (remove_first (remove_last q)) ([first q; last q]::acc)
    else
      sadzaj (remove_last q) ([last q]::acc)
  in
    sadzaj (make_queue l) [];;

kajaki [1; 2; 2; 3; 3; 3; 4; 6; 6; 8; 8; 8; 9; 9; 10] 10;;
[[3; 4]; [3; 6]; [3; 6]; [8]; [2; 8]; [2; 8]; [9]; [1; 9]; [10]]
```

Uzasadnienie poprawności.

To rozwiązanie generuje takie same wyniki jak poprzednie (z dokładnością do kolejności elementów na liście). Dlaczego?

4. Możemy też wsadzać do kajaka parę dwóch kolejnych, jak najgrubszych. Grubi są ci, którzy nie zmieszczą się do jednego kajaka razem z kolejnym chudszym. Najchudszy jest z definicji chudy. Sadzamy razem dwóch najgrubszych chudych. Stopniowo grubi mogą przenosić się do puli chudych.

   Tym razem zamiast kolejek wystarczą nam listy. Lista grubych jest uporządkowana niemalejąco, a chudych nierosnąco.

```
let kajaki l wyp =
  let rec dobierz ch g =
    match (ch, g) with
      (_, []) -> (ch, []) |
      ([], h::t) -> dobierz [h] t |
      (chh::cht, gh::gt) ->
        if chh + gh <= wyp then
          dobierz (gh::ch) gt
        else
          (ch, g)
  in
    let rec sadzaj chp gp acc =
      let (ch, g) = dobierz chp gp
      in
        match ch with
          [] -> acc |
          [h] -> sadzaj [] g ([h]::acc) |
          h1::h2::t -> sadzaj t g ([h2; h1]::acc)
    in
      sadzaj [] l [];;

kajaki [1; 2; 2; 3; 3; 3; 4; 6; 6; 8; 8; 8; 9; 9; 10] 10;;
[[10]; [9]; [9]; [1; 8]; [2; 8]; [2; 8]; [3; 3]; [3; 6]; [4; 6]]
```

Uzasadnienie poprawności.

Ćwiczenia

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

1. [II OI, Palindromy] Podziel dane słowo (listę znaków) na maksymalną liczbę palindromów parzystej długości.

2. [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. Czy jest to zbiór o minimalnej liczbie elementów?

3. [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.

   - wersja łatwiejsza: między dwoma miastami może być wiele połączeń,
   - wersja trudniejsza: między dwoma miastami może być co najwyżej jedno połączenie.

4. [Wektorki 1D, podobne do zadania Steps z Valladolid] Tworzymy ciągi liczb całkowitych (x_k) i (v_k) w następujący sposób:

   - $x_0 = 1$, $v_0 = 0$,
   - dla każdego $k > 0$ wybieramy pewne v_k spełniające $v_{k-1} - 1 \leq v_k \leq v_{k-1} + 1$, oraz przyjmujemy $x_k = x_{k-1} + v_k$.

   Napisz procedurę `wyścig: int` $\rightarrow$ `int list`, która dla danego $n \geq 1$ znajdzie **najkrótszy** taki ciąg $x_0, x_1, ..., x_k$, który spełnia powyższe warunki, oraz $x_k = n$. Wynikiem procedury powinna być lista $[x_0; ...; x_k]$.

5. [X OI] Dana jest tabliczka czekolady, którą należy połamać na cząstki. Każde przełamanie kawałka czekolady jest obarczone pewnym kosztem. Koszt ten nie zależy od wielkości kawałka czekolady, ale od prostej, wzdłuż której łamiemy. Dla wszystkich prostych, na których leżą krawędzie cząstek czekolady mamy stałe określające koszty łamania.

   (Można udowodnić, że istnieje rozwiązanie optymalne, w którym łamiemy kawałki przez całą szerokość tabliczki czekolady.)

6. [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.)

7. [II OI] Jak obejść wszystkie wierzchołki drzewa (ogólnego) tak, aby odległość między kolejnymi wierzchołkami nie przekraczała 3.

8. [XI OI] Dana jest grupa osób, które chcą przejść nocą przez most. Mają oni jedną latarkę, która pozwala przejść jednej lub dwóm osobom. Każda osoba potrzebuje określonego

czasu na przejście przez most. (Możesz założyć, że dane są posortowane.) Jeśli dwie osoby idą razem, to potrzebują tyle czasu, co wolniejsza z nich. Jaki jest minimalny czas potrzebny do przejścia wszystkich?

9. [XII OI, OSC] Jasio jest trzylatkiem i bardzo lubi bawić się samochodzikami. Jasio ma n różnych samochodzików. Wszystkie samochodziki leżą na wysokiej półce, do której Jasio nie dosięga. W pokoju jest mało miejsca, więc na podłodze może znajdować się jednocześnie co najwyżej k samochodzików.

   Jasio bawi się jednym z samochodzików leżących na podłodze. Oprócz Jasia w pokoju cały czas przebywa mama. Gdy Jasio chce się bawić jakimś innym samochodzikiem i jeśli ten samochodzik jest na podłodze, to sięga po niego. Jeśli jednak samochodzik znajduje się na półce, to musi mu go podać mama. Mama podając Jasiowi jeden samochodzik, może przy okazji zabrać z podłogi dowolnie wybrany przez siebie inny samochodzik i odłożyć go na półkę (tak, aby na podłodze nie brakło miejsca).

   Mama bardzo dobrze zna swoje dziecko i wie dokładnie, którymi samochodzikami Jasio będzie chciał się bawić. Dysponując tą wiedzą mama chce zminimalizować liczbę przypadków, gdy musi podawać Jasiowi samochodzik z półki. W tym celu musi bardzo rozważnie odkładać samochodziki na półkę. (Za tą historyjką kryje się problem optymalnej strategii wymiany stron w pamięci wirtualnej.)

   Napisz procedurę `samochodziki`, która dla danej liczby k oraz listy samochodzików, którymi zechce się bawić Jasio, obliczy minimalną liczbę razy, kiedy mama będzie musiała Jasiowi podać samochodzik.

Wytyczne dla prowadzących ćwiczenia

Ad. 1 Oczekujemy złożoności czasowej $O(n^2)$.

Laboratorium

Sortowanie topologiczne. Termin: 2 tygodnie, 3 punkty.
Budowa domu składa się z wielu etapów. Niektóre z nich w sposób oczywisty muszą być wykonane przed innymi, ale czasem nie jest oczywiste, czy jeden etap musi być wykonany przed drugim, czy nie. Na przykład, najpierw kładziemy glazurę i terakotę, a potem parkiet. Ale kiedy możemy kłaść gładź gipsową na ściany? Czy można to zrobić przed położeniem glazury, czy może trzeba poczekać aż będzie położony parkiet?

Mamy n czynności ponumerowanych od 1 do n. Dana jest lista par postaci $l = [(p_1, q_1); \ldots; (p_k, q_k)]$. Każda taka para oznacza, że czynność p_i musi być wykonana przed czynnością q_i. Kolejność par na liście nie ma znaczenia. Napisz procedurę `tsort : int -> (int * int) list -> int list`, która wywołana jako `tsort` n l obliczy przykładową kolejność wykonania wszystkich n czynności.
Podpowiedzi:

- Istnieją przynajmniej dwa, istotnie różne rozwiązania, oba o optymalnej złożoności.
- Jedna możliwość jest taka, że utrzymujemy kolekcję czynności, a dla każdej czynności mamy liczbę czynności, które mają być wykonane przed nią i listę czynności, które mają być wykonane po niej.
- Czynności przed którymi nic nie musi być wykonane mogą być wykonane jako pierwsze i można je usunąć ze struktury danych.
- Inny pomysł polega na tym, że przed daną czynnością musimy wykonać wszystkie czynności, które musimy przed nią wykonać, i takie, które musimy przed nimi wykonać itd.

Wykład 21. Algorytm mini-max

Problem: jak zaprogramować optymalną strategię grania w grę dla dwóch graczy? Możemy tu zastosować technikę podobną do back-trackingu, 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 ponadto może być większa lub mniejsza. Może to odpowiadać liczbie wygranych punktów i/lub pieniędzy. Sytuacje te możemy ocenić za pomocą funkcji przypisującej im wagi rzeczywiste. Wagi dodatnie oznaczają wygraną jednego gracza, wagi ujemne drugiego, a zero to remis.

W ten sposób 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.

21.1 Algorytm mini-max

Sam algorytm postaramy się zapisać w sposób niezależny od rodzaju gry. Graczy będziemy reprezentować przez liczby $+1$ i -1, w zależności od tego, czy grają na max, czy na min.

```
let i = 1;;

let other who = -who;;

let rec minimax s who =
```

```
  if finished s who then
    (result s who, pass)
  else
    let make_move m =
      let (w, _) = minimax (move s who m) (other who)
      in (who * w, m)
    in
      let (w, m) = choose (map make_move (moves s who))
      in (w * who, m);;
```

Korzystamy tu z pomocniczej procedury wybierającej optymalny ruch:

```
let choose l =
  match l with
    [] -> failwith "Pusta lista ruchów!" |
    (h::t) ->
      fold_left
        (fun (w, m) (w1, m1) -> if w1 > w then (w1, m1) else (w, m))
        h t;;

let game s =
  let (_, m) = minimax s 1
  in m;;
```

21.2 Przykład — gra w kamienie

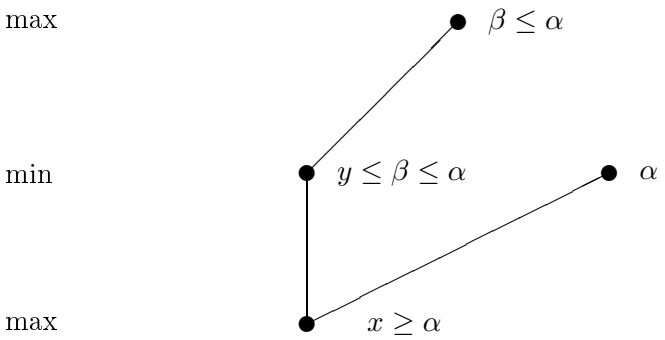
Zilustrujmy na przykładzie prostej gry w kamienie brakujące elementy. Każdy z graczy usuwa z kupki 1–3 kamieni. Wygrywa ten, który wykona ostatni ruch.

```
let moves s who =
  match s with
    0 -> [] |
    1 -> [1] |
    2 -> [1; 2] |
    _ -> [1; 2; 3];;

let finished s who = moves s who = [];;

let pass = 0;;

let result s who = other who;;

let move s who m = s - m;;
```

21.3 α-β obcięcie

[[Uwzględnić wpływ α o 4 poziomy i głębiej.]] 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.

```
exception Obciecie of int * move;;

let rec alfabeta s who alpha =
  if finished s who then (result s who, pass) else
    let choose mvs =
      try
        fold_left
          (fun (beta, mb) m ->
            let (w, _) = alfabeta (move s who m) (other who) beta
            in
              if w * who >= alpha * who then raise (Obciecie (w, m)) else
```

```
            if w * who > beta * who then (w, m) else (beta, mb))
        (no_result who, pass)
        mvs
    with Obciecie (beta, mb) -> (beta, mb)
  in
    choose (moves s who);;

let game s =
  let (_, m) = alfabeta s i (no_result (other i))
  in m;;
```

Korzystamy tu z pewnych dodatkowej procedury zależnej od gry:

```
let no_result who = -2 * who;;
```

Dalsze usprawnienie można uzyskać poprzez dodanie spamiętywania. Często ta sama sytuacja w grze może być osiągnięta na wiele sposobów. Oznacza to, że to samo poddrzewo może pojawiać się wielokrotnie w grze. Zamiast wielokrotnie przeprowadzać dla niego te same obliczenia, stosujemy technikę spamiętywania:

```
let tab = ref empty;;

let rec alfabeta s who alpha =
  memoize tab (function (s, who, alpha) ->
    ...
```

W przypadku gry w kamienie, spamiętywanie ogranicza koszt do liniowego ze względu na liczbę kamieni.

21.4 Analiza algorytmu α-β obcięcia

Efektywność tego algorytmu istotnie zależy od kolejności rozpatrywanych ruchów. Ruchy należy, w miarę możliwości, starać się rozpatrywać od lepszych do gorszych. W ten sposób większa część gałęzi jest obcinana. Z braku innych kryteriów możemy też wybierać najpierw ruchy prowadzące szybciej do zakończenia rozgrywki. W ten sposób mamy większe szanse na obcięcie gałęzi których przejrzenie jest bardzo pracochłonne.

Zastanówmy się jednak, jakiego rzędu oszczędności może przynieść α-β obcięcie w porównaniu z algorytmem mini-max. Zauważmy, że na każdym poziomie drzewa gry, pierwsze wywołanie rekurencyjne dokona obliczenia wszystkich podgałęzi. Natomiast w kolejnych wywołaniach rekurencyjnych mamy coraz większe szanse na obcinanie gałęzi.

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 $B_1, \ldots, B_n$. Załóżmy, że prawdopodobieństwo, iż $B_i > A$ jest równe p (niezależnie od A oraz i). Wówczas prawdopodobieństwo, że będziemy obliczać wagę i gałęzi wynosi:

- $p^{i-1}(1-p)$, dla $i < n$, oraz
- p^{n-1}, dla $i = n$.

Tak więc oczekiwana liczba obliczanych wag gałęzi wynosi:

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

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

Zauważmy, że oszacowanie to nie zależy od n i np. dla $p = \frac{1}{2}$ otrzymujemy średnio dwa wywołania rekurencyjne.

Możemy dalej oszacować liczbę wywołań rekurencyjnych na poziomie drugim:

$$n + \frac{n-1}{1-p} = n \left(1 + \frac{1}{1-p} \right) - \frac{1}{1-p} \leq n \frac{2-p}{1-p}$$

Biorąc pod uwagę wykładniczy rozrost drzewa przekłada się to na zmianę stopni wierzchołków w drzewie z n na $\sqrt{\frac{2-p}{1-p} n}$, co np. dla $p = \frac{1}{2}$ daje nam $\sqrt{3n}$.

Oczywiście powyższe oszacowanie jest zgrubne. Algorytm α-β obcięcia nigdy nie wykona więcej kroków niż algorytm mini-max, potrafi jednak istotnie zmniejszyć średnią liczbę wywołań rekurencyjnych, a co za tym zmienić istotnie asymptotyczną złożoność programu.

Ćwiczenia

Rozwiązanie każdego z poniższych zadań wymaga użycia jednej z trzech technik: back-trackingu, programowania dynamicznego lub zachłannego. Pytanie której?

- Zadanie o misjonarzach i kanibalach. Przez rzekę chcą przeprawić się kanibale i misjonarze. Kanibali i misjonarzy jest po trzech. Mają jedną łódkę, którą może płynąć do dwóch osób. Łódką umieją wiosłować kanibale i tylko jeden misjonarz. Jeżeli w dowolnej chwili, na dowolnym brzegu rzeki będzie więcej kanibali, niż misjonarzy, to zjedzą oni misjonarzy. W jaki sposób wszyscy mogą bezpiecznie przeprawić się na drugi brzeg.
- [CLR] Minimalny koszt mnożenia macierzy.
- [CLR 16.3] Dane są dwa napisy (listy liter). Dostępne są operacje:
  - przepisz pierwszą literę z pierwszego ciągu na koniec drugiego,
  - zamień pierwszą literę pierwszego ciągu na inną literę i przenieś ją na koniec drugiego ciągu,
  - wstaw na koniec drugiego ciągu literę,
  - usuń z początku pierwszego ciągu pierwszą literę.

  Każda z tych operacji ma dodatni koszt. Jaki jest minimalny koszt zamiany pierwszego ciągu w drugi?
- [CLR 17.2-5] Dany jest zbiór punktów na osi. Podaj minimalną liczbę odcinków jednostkowych, jakimi można pokryć punkty z tego zbioru.
- Rozpatrujemy przedstawienia liczby naturalnej n jako sumy różnych składników nieparzystych, tzn. rozważamy takie ciągi $(x_1, \ldots, x_k)$, że:
  - $1 \leq x_i$ i x_i jest nieparzyste (dla $i = 1, \ldots, k$),
  - $x_1 > x_2 > \cdots > x_k$,
  - $x_1 + \cdots + x_k = n$.

  - Napisz procedurę, która dla danej liczby n obliczy, na ile różnych sposobów można ją przedstawić jako sumę różnych nieparzystych składników. Na przykład, dla $n = 2$ poprawnym wynikiem jest 0, a dla $n = 12$ poprawnym wynikiem jest 3.
  - Napisz procedurę, która dla danej liczby n wyznaczy (pewne) przedstawienie tej liczby jako sumy maksymalnej liczby nieparzystych składników. Na przykład, dla $n = 42$ poprawnym wynikiem może być `[15; 11; 7; 5; 3; 1]`.
- Tworzymy ciąg liczb całkowitych (x_k) w następujący sposób:
  - $x_0 = 1$,
  - dla każdego $k > 0$ wybieramy pewne $0 \leq i, j < k$ i przyjmujemy $x_k = x_i + x_j$.

  Napisz procedurę `ciąg: int` $\rightarrow$ `int list`, która dla danego $n \geq 1$ znajdzie **najkrótszy** taki ciąg $x_0, x_1, ..., x_k$, który spełnia powyższe warunki, oraz $x_k = n$. Wynikiem procedury powinna być lista $[x_0; ...; x_k]$.

  Rozwiązując to zadanie można łatwo wybrać złą technikę. Konieczne jest więc solidne uzasadnienie poprawności. Oto garść kontrprzykładów, które należy wziąć pod uwagę:

n	Wynik
15	1, 2, 3, 6, 12, 15
382	1, 2, 4, 5, 9, 14, 23, 46, 92, 184, 198, 382
77	1, 2, 4, 5, 9, 18, 36, 41, 77
12509	1, 2, 3, 6, 12, 13, 24, 48, 96, 192, 384, 397, 781, 1562, 3124, 6248, 6261, 12509

- [Bentley, CLR 16-1] Bitoniczne rozwiązanie problemu komiwojażera to taki cykl w kształcie wielokąta, który począwszy od skrajnie lewego wierzchołka, idzie w prawo (tworząc górny obrys wielokąta) do skrajnie prawego, po czym wraca do skrajnie lewego (tworząc obrys dolny). Wyznacz najkrótszy bitoniczny cykl Hamiltona dla danego zbioru punktów na płaszczyźnie.

- [II OI] Dany jest zestaw n czynności. Wykonanie każdej z tych czynności trwa tym dłużej, im później się ją zacznie, konkretnie jeżeli zaczniemy wykonywać czynność i w chwili t, to jej wykonanie będzie trwać $a_i t + b_i$. Jaki jest minimalny czas wykonania wszystkich tych czynności, jeżeli zaczynamy je wykonywać w chwili 0?

- Dana jest drzewo typu:

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

  opisujące kształt choinki.

  1. [PCh] W węzłach choinki (`Rozgalezienie`, `Galaz` i `Koniuszek`) możemy umieszczać świeczki. Powiemy, że choinka jest oświetlona, jeżeli dla każdej krawędzi (łączącej węzły) choinki przynajmniej w jednym jej końcu znajduje się świeczka. Napisz procedurę, która dla danej choinki obliczy minimalną liczbę świeczek potrzebnych do oświetlenia choinki.
  2. [PCh] Na każdej krawędzi (łączącej węzły) choinki wieszamy jedną bombkę. Mamy do dyspozycji trzy rodzaje bombek: czerwone po 2 zł, niebieskie po 5 zł i srebrne po 7 zł. Choinkę należy przystroić w taki sposób, żeby na krawędziach o końcach w tym samym węźle wisiały bombki o różnych kolorach, a łączny koszt przystrojenia całej choinki był jak najmniejszy.
  3. W węzłach choinki (`Rozgalezienie`, `Galaz` i `Koniuszek`) wieszamy bombki. Dostępne są trzy rodzaje bombek:
     - czerwone, wykonane z miedzi, każda taka bombka waży 1 kg,
     - srebrne, wykonane z białego złota niskiej próby, każda taka bombka waży 2 kg,
     - złote, wykonane ze szczerego złota, każda taka bombka waży 3 kg.

     Bombki należy zawiesić tak, aby choinka nie chwiała się, tzn. dla każdego rozgałęzienia łączny ciężar bombek zawieszonych po lewej i po prawej stronie musi być taki sam. Pytanie czy jest to możliwe, a jeżeli tak, to na ile sposobów?

     Napisz procedurę, która dla danej choinki obliczy na ile sposobów można powiesić na niej bombki.

Wykład 22. Gry typu wygrana/przegrana i drzewa AND-OR

22.1 Drzewa AND-OR

W niniejszym wykładzie przedstawimy materiał dotyczący gier kończących się wynikiem binarnym — wygrana/przegrana. W przypadku takich gier wagi przyporządkowywane węzłom drzewa mini-max mogą przyjmować tylko dwie wartości. Jeśli przyjmiemy, że naszą wygraną oznaczymy przez `true`, a przeciwnika przez `false`, to zamiast mówić o maksimum i minimum możemy powiedzieć, że my gramy na `or`, a przeciwnik na `and` — stąd nazwa: drzewa AND-OR.

Rozpatrujemy częściowo rozwinięte drzewo gry. Wierzchołki drzewa są typu AND lub OR, w zależności od tego kto w danej sytuacji ma wykonać ruch. Liście mogą reprezentować sytuacje końcowe, lub sytuacje, w których gra może się dalej rozwijać. Liście reprezentujące sytuacje końcowe mają przyporządkowane wagi zgodnie z tym kto wygrał. Liście reprezentujące sytuacje nierozstrzygnięte mają wagi nieokreślone. Liście wewnętrzne mają określone wagi, jeśli można je określić na podstawie wag ich potomków.

Jeśli korzeń drzewa ma wyznaczoną wagę, to określa ona, czy mamy strategię wygrywającą, a wagi synów korzenia wskazują, jaki ruch należy wykonać. Jeżeli korzeń drzewa nie ma określonej wagi, to należy rozwinąć drzewo. Pytanie tylko w którym miejscu?

22.2 Zbiory falsyfikujące/potwierdzające

Załóżmy, że korzeń ma wagę nieokreśloną.

Fakt: Nadając liściom z niekreślonymi wagami wagi `true` możemy doprowadzić do tego, że korzeń ma wagę `true`, i odwrotnie, nadając liściom wagi `false` możemy doprowadzić do tego, że korzeń będzie miał wagę `false`.

Dowód: Bo operacje `and` i `or` są monotoniczne.

Definicja: Zbiór *falsyfikujący* (*potwierdzający*) to taki zbiór liści o nieokreślonych wagach, który jeżeli liście te uzyskają wagi `false` (`true`), to korzeń uzyska też wagę `false` (`true`).

Fakt: Dowolne dwa zbiory potwierdzający i falsyfikujący mają niepuste przecięcie.

Dowód: Gdyby tak nie było, to moglibyśmy przypisać liściom z jednego zbioru wagi `true`, a z drugiego `false` i korzeniowi można by przypisać równocześnie obie wagi — sprzeczność.

Interesują nas **minimalne** zbiory falsyfikujące i potwierdzające. Jeżeli określimy wartość wierzchołka należącego do przecięcia zbioru falsyfikującego i potwierdzającego, to wyeliminujemy jeden z nich. Strategia polega na tym, aby cyklicznie rozwijać wierzchołek, który należy do przecięcia zbiorów falsyfikującego i potwierdzającego o minimalnej liczbie elementów. Szukany wierzchołek możemy wyznaczyć obchodząc drzewo i korzystając z następujących faktów:

Fakt:

- Jeżeli korzeń drzewa jest typu `and`, to:
  - najmniejszy zbiór falsyfikujący, to zbiór falsyfikujący dla jednego z synów,

  - najmniejszy zbiór potwierdzający, to suma najmniejszych zbiorów potwierdzających dla synów,
  - szukany element jest w tym poddrzewie, które ma najmniejszy zbiór falsyfikujący.

- Jeżeli korzeń drzewa jest typu `or`, to dualnie.

Rozwijamy tak drzewo gry, aż określimy wartość korzenia.

22.3 Zwinięcie drzewa do DAG-u

Opisany algorytm możemy usprawnić nie konstruując drzewa, lecz sklejając wierzchołki odpowiadające identycznym sytuacjom na planszy. W ten sposób uzyskujemy DAG. Problem wyznaczenia najmniejszego zbioru falsyfikującego/potwierdzającego dla DAG-u jest trudny. Stosuje się więc następującą heurystykę: wybieramy do rozwinięcia taki wierzchołek, jakbyśmy mieli drzewo stanowiące rozwinięcie DAGu. Oczywiście rozwijając DAG należy pilnować, aby sklejać powtarzające się wierzchołki. Możemy tu stosować spamiętywanie sytuacji, dla których utworzyliśmy wierzchołki.

22.4 Implementacja

Definicje wstępne, niezależne od gry:

```
type player = And | Or;;
let i = Or;;

let other = function And -> Or | Or -> And;;

type boolean = True | False | Undefined;;

let and3 x y =
  match (x, y) with
    (False, _) -> False |
    (_, False) -> False |
    (True, _)  -> y |
    (_, True)  -> x |
    _          -> Undefined;;

let or3 x y =
  match (x, y) with
    (True, _)  -> True |
    (_, True)  -> True |
    (False, _) -> y |
    (_, False) -> x |
    _          -> Undefined;;

let won = True;;
let lost = False;;
```

Zakładamy, że gra jest określona podobnie, jak gra w kamienie w poprzednich wykładach.

```
(* Drzewo AND-OR jest reprezentowane za pomocą mapy przyporządkowującej *)
(* parom (sytuacja, gracz) n-tki postaci:                                *)
(* - wartość: True / False / Undefined                                   *)
(* - minimalna liczba el. falsyfikujących,                               *)
(* - minimalna liczba el. potwierdzających,                              *)
(* - dla rozwiniętych: lista trójek (ruch, sytuacja, gracz).             *)


(* Selektory *)
let vertex dag s p = apply dag (s, p);;

let value (v, _, _, _) = v;;
let min_f (_, f, _, _) = f;;
let min_t (_, _, t, _) = t;;
let succ  (_, _, _, s) = s;;

let succ_move  (m, _, _)   = m;;
let succ_board (_, b, _)   = b;;
let succ_who   (_, _, who) = who;;
let follow dag x = vertex dag (succ_board x) (succ_who x);;

(* Węzeł jest rozwinięty, jeżeli ma synów lub ma wartość. *)
let is_expanded v = (value v <> Undefined) || (succ v <> []);;


(* Operacje na DAG-u *)

(* Czy już jest taki wierzchołek? *)
let is_present dag s who = dom dag (s, who);;

(* Dodanie do DAG-u węzła o parametrach:         *)
(* - s   - sytuacja,                             *)
(* - who - gracz,                                *)
(* - v   - wartosc,                              *)
(* - e   - lista trójek (ruch, sytuacja, gracz). *)
let insert dag s who v e =

  (* Rozmiar najmniejszego zbioru falsyfikującego/potwierdzającego *)
  (* (w zależności od parametrów selector/cumulator) dla synów.    *)
  let minft selector cumulator =
    (*  Lista liczb do przeliczenia, pochodzących od  *)
    (* synów z nieokreślonymi wartościami. *)
    let minl = map selector
                (filter
                   (fun x -> value x = Undefined)
                   (map (follow dag) e))
    in
```

```
      if e = [] && v = Undefined then
        (* Wierzchołek nierozwinięty. *)
        1
      else if minl <> [] then
        (* Zliczenie ilości wierchołków wyznaczających wartość. *)
        cumulator minl
      else
        (* Wszyscy synowie są określeni. *)
        infty

  (* Przelicz wartość wierzchołka. *)
  and recalculate_value =
    if v = Undefined && e <> [] then
      let ss = map (fun x -> value (follow dag x)) e
      in
        if who = And then
          fold_left and3 True ss
        else
          fold_left or3 False ss
    else v
  in
    (* Liczba wierzchołków falsyfikujących. *)
    let minf = minft min_f (if who = Or then sum else minl)

    (* Liczba wierzchołków potwierdzających. *)
    and mint = minft min_t (if who = And then sum else minl)
    in

      update dag (s, who) (recalculate_value, minf, mint, e);;


(* Dodanie nierozwiniętego liścia. *)
let insert_leaf dag s who =
  if is_present dag s who then
    dag
  else
    insert dag s who Undefined [];;

(* Początkowa sytuacja do rozważenia. *)
let initial s = insert_leaf empty s i;;


(* Rozwija w danym dag-u węzeł v odpowiadający sytuacji s dla gracza who. *)
let rec expand_dag dag s who =

  (* Rozwija liść v odpowiadający sytuacji s. *)
  let expand_leaf v =
```

```
  if finished s who then begin
    (* Rozgrywka skończona. *)
    insert dag s who (result s who) []
  end else
    (* Rozwiń możliwe ruchy. *)
    let leafs = map (fun m -> (m, move s who m)) (moves s who)
    in
      (* Wstaw nowych synów do DAG-u. *)
      let dag1 =
        fold_left
          (fun d (m, s) -> insert_leaf d s (other who))
          dag leafs
      (* Lista następników. *)
      and e = map (fun (m, s) -> (m, s, other who)) leafs
      in
        insert dag1 s who (value v) e

(* Przelicza wartość węzła na podstawie wartości potomków. *)
and recalculate_value d v =
  let ss = map (fun x -> value (follow d x)) (succ v)
  in
    let vv =
      if who = And then
        fold_left and3 True ss
      else
        fold_left or3 False ss
    in
      insert d s who vv (succ v)
in

  (* Rozwija podgraf zaczepiony w węźle v wybierając minimalną *)
  (* wartość podanego aspektu.                                 *)
  let expand_compound v aspect =
    (* Predykat sprawdzający, czy syn wymaga rozwinięcia. *)
    let test s =
      let vv = follow dag s
      in aspect vv = aspect v && value vv = Undefined
    in
      (* Rozwijamy syna realizującego min-f/min-t *)
      let ssl = filter test (succ v)
      in
        if ssl = [] then
          (* Potomek jest już rozwinięty. *)
          recalculate_value dag v
        else
          let dag1 = expand_dag dag (succ_board (hd ssl)) (other who)
          in
```

```
            recalculate_value dag1 v
    in
      let v = vertex dag s who
      in
        if not (is_expanded v) then
          (* Rozwinięcie liścia. *)
          expand_leaf v
        else if succ v = [] then
          (* Nie ma co rozwijać. *)
          dag
        else if who = And then
          (* Rozwinięcie węzła And. *)
          expand_compound v min_f
        else
          (* Rozwinięcie węzła Or. *)
          expand_compound v min_t;;


(* Rozwijaj, aż zostanie określona waga korzenia. *)
let rec keep_expanding dag s who =
  if value (vertex dag s who) = Undefined then
    keep_expanding (expand_dag dag s who) s who
  else
    dag;;

(* Najlepszy ruch. *)
let game s =
  let dag = initial s
  in
    let dag1 = keep_expanding dag s i
    in
      let v = vertex dag1 s i
      in
        (succ_move (hd (filter
                          (fun s -> value v = value (follow dag1 s))
                          (succ v))));;
```

Wykład 23. Wyszukiwanie wzorców i algorytm Knutha-Morrisa-Pratta

Zastanówmy się nad następującym problemem. Mamy dane dwa teksty, szukamy wystąpień pierwszego tekstu w drugim. Na nasze potrzeby przyjmujemy, że tekst to lista symboli. Wynikiem powinna być lista pozycji wszystkich wystąpień wzorca w tekście.

23.1 Naiwny algorytm wyszukiwania wzorca

Algorytm naiwny sprawdza wszystkie możliwe pozycje, na których wzorzec występuje w tekście i sprawdza, czy faktycznie tak jest.

```
let rec prefix w t =
  match w with
    [] -> true |
    hw::tw ->
      match t with
        [] -> false |
        ht::tt -> (hw = ht) && prefix tw tt;;

let pattern_match w t =
  let rec iter t n acc =
    let a = if prefix w t then n::acc else acc
    in
      match t with
        [] -> rev a |
        _::tail -> iter tail (n+1) a
  in
    iter t 1 [];;
```

Złożoność czasowa tego algorytmu jest rzędu $\Theta(|w| \cdot |t|)$. Czy można lepiej? Oczywiście tak.

23.2 Algorytm Rabina-Karpa

[[Uzupełnić, CLR p.34.2]]

23.3 Algorytm KMP

Algorytm przedstawiony poniżej pozwala wyznaczyć wszystkie wystąpienia wzorca w tekście w czasie liniowym. Jego zastosowanie jest jednak trochę szersze. Dokładniej, algorytm ten dla zadanego tekstu wyznacza wartości tzw. funkcji prefiksowej P. Dla danego tekstu $t = (t_1 t_2 \dots\ t_n)$ funkcja prefiksowa jest zdefiniowana następująco.

$P(i)$ to największe takie j, że:

- $0 \leq j < i$,
- $(t_1 \dots\ t_j) = (t_{i-j+1} \dots\ t_i)$.

Inaczej mówiąc, $P(i)$ to długość najdłuższego właściwego prefiksu $(t_1 t_2 \dots\ t_i)$, który jest równocześnie sufiksem $(t_1 t_2 \dots\ t_i)$.

Oznaczmy dodatkowo przez $G(i)$ zbiór

$$G(i) = \{0 \leq j < i : (t_1 \dots\ t_j) = (t_{i-j+1} \dots\ t_i)\}$$

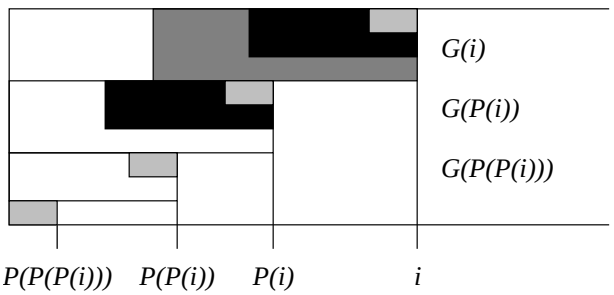
Czyli $G(i)$ to zbiór długości wszystkich takich właściwych prefiksów $(t_1 t_2 \dots\ t_i)$, które są równocześnie sufiksami tego słowa. Dodatkowo przyjmujemy, że:

$$G(0) = \emptyset$$

$$P(0) = 0$$

Zachodzą następujące własności:

- $P(i) = \max G(i)$,
- $P(1) = 0$,
- $P(i) \geq 0$,
- $P(i) < i$ (dla $i > 0$),
- $i > P(i) > P(P(i)) > \dots > P^k(i) = 0$, dla pewnego $k \in \mathbb{N}$,
- $G(i+1) = \{0\} \cup \{j+1 : j \in G(i) \wedge t_{i+1} = t_{j+1}\}$,
- $G(i) = \{P(i)\} \cup G(P(i))$ — największy element w $G(i)$ to właśnie $P(i)$, natomiast każdy mniejszy element $G(i)$ to długość prefiksu tekstu, który jest sufiksem $(t_1 t_2 \dots\ t_{P(i)})$.



- $G(i) = \{P^j(i) : 1 \leq j \leq k\}$,
- $G(i+1) = \{0\} \cup \{P^j(i) + 1 : 1 \leq j \leq k \wedge t_{i+1} = t_{P^j(i)+1}\}$,
- $P(i+1) = \max(\{0\} \cup \{P^j(i) + 1 : 1 \leq j \leq k \wedge t_{i+1} = t_{P^j(i)+1}\})$,
- dla $i > 0$ mamy $P(i+1) = \begin{cases} P^j(i)+1 & ; \text{ gdzie } j \text{ jest najmniejsze takie, że } 1 \leq j \leq k \wedge t_{i+1} = t_{P^j(i)+1} \\ 0 & ; \text{ wpp.} \end{cases}$,

A oto i implementacja:

```
let kmp t =
  let
    p = make (length(t) + 1) 0 and
    pj = ref 0
  in begin
    for i = 2 to length t do
      while (!pj > 0) && (t.(!pj) <> t.(i - 1)) do
        pj := p.(!pj)
      done;
      if t.(!pj) = t.(i - 1) then pj := !pj + 1;
      p.(i) <- !pj
    done;
    p
  end;;
```

Chcąc zastosować algorytm KMP do wyszukiwania wzorców obliczamy tablicę prefiksową dla szukanego wzorca. Z jej pomocą możemy znaleźć wystąpienia wzorca w czasie liniowym.

```
let find x y =
  let
    i = ref 0 and
    j = ref 0 and
    w = ref [] and
    p = kmp x
  in
    while !i <= length y - length x do
      j := p.(!j);
      while (!j < length x) && (x.(!j) = y.(!i + !j)) do
        j := !j + 1
      done;
      if !j = length x  then w := !i::!w;
      i := !i + if !j > 0 then !j - p.(!j) else 1
    done;
    rev !w;;

find [|'a'; 'l'; 'a'|] [|'a'; 'l'; 'a'; 'l'; 'a'; 'b'; 'a'; 'm'; 'a'|];;
[0; 2]
```

23.4 Zastosowanie tablicy prefiksowej w automatach skończonych

[[Uzupełnić]]

[[Zastanowić się nad dodaniem algorytmu Boyera-Moora.]]

Ćwiczenia

- Dana jest lista elementów. Wyznacz wszystkie nietrywialne rotacje cykliczne danej listy, które dają w wyniku ją samą.
- Niech $(x_1, x_2, \ldots, x_n)$ będzie danym niepustym ciągiem liczb. Okresem tego ciągu nazwiemy najmniejszą taką liczbę k, że:
  - $1 \leq k \leq n$,
  - k jest dzielnikiem n,
  - $x_i = x_{i+k}$ dla wszystkich $i = 1, \ldots n - k$.

  Zauważmy, że każdy ciąg ma okres, co najwyżej jest to $k = n$. Na przykład, okresem ciągu $[1; 3; 2; 1; 1; 3; 2; 1; 1; 3; 2; 1]$ jest 4, a okresem ciągu $[1; 3; 2; 1; 1]$ jest 5.

  Napisz procedurę `okres: int array -> int`, która dla danego ciągu wyznaczy jego okres.
- [XII OI] Chcemy wykonać długi napis. W tym celu najpierw przygotowujemy odpowiedni szablon z wyciętymi literkami. Następnie taki szablon można przyłożyć we właściwym miejscu do ściany i malujemy po nim farbą. Malujemy zawsze po całym szablonie, ale dopuszczamy, aby litery były malowane wielokrotnie. Literki na szablonie znajdują się koło siebie (nie ma tam przerw). Zadanie polega na wyznaczeniu minimalnej długości szablonu, za pomocą którego można wykonać cały napis.
- [XIII OI] Napis Q jest okresem A, jeśli Q jest prefiksem właściwym A oraz A jest prefiksem (niekoniecznie właściwym) napisu QQ. Przykładowo, napisy *abab* i *ababab* są okresami napisu *abababa*. Maksymalnym okresem napisu A nazywamy najdłuższy z jego okresów, lub napis pusty, jeśli A nie posiada okresu. Dla przykładu, maksymalnym okresem napisu *ababab* jest *abab*. Maksymalnym okresem *abc* jest napis pusty.

  Napisz procedurę, która dla danego (w postaci listy) napisu obliczy sumę długości maksymalnych okresów wszystkich jego prefiksów.

Laboratorium

Należy napisać program, który wczyta ze standardowego wejścia ciąg słów pooddzielanych białymi znakami (przez słowo rozumiemy maksymalny ciąg znaków nie zawierający białych znaków) a następnie wypisze wszystkie klasy abstrakcji relacji anagram określonej na tych słowach.

Dwa słowa 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 osobnym wierszu.

Przykładowy wynik pracy programu Dla danych:

```
ala  baba

   abba ala
lala
```

wynik powinien wyglądać następująco:

```
abba baba
ala
lala
```

Założenia dotyczące implementacji Program powinien zawierać wyodrębniony plik (lub moduł): `klasy` — implementujący operacje na strukturze danych przechowującej klasy abstrakcji. Moduł ten powinien dostarczać następujące operacje:

- `puste` — inicjuje pusty zbiór klas abstrakcji,
- `wstaw_słowo` — wstawia słowo do odpowiedniej klasy abstrakcji (ew. ja inicjując),
- `klasy` — podaje listę list elementów klas.

Ponadto powinien być wydzielony moduł wczytujący dane i dzielący je na słowa. Moduł ten powinien udostępniać (m.in.) procedurę `podaj_słowo` [[Doprecyzować w postaci sygnatury.]]

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

Wykład 24. Przeszukiwanie grafów

[[Definicje podstawowych pojęć: Grafu skierowanego i nieskierowanego, ścieżki, ścieżki prostej, cyklu, cyklu prostego, spójnej składowej, odległości między wierzchołkami. ...]]

24.1 Grafy i problem przeszukiwania grafów

Definicja grafu (nieskierowanego).

[[Zdecydować się, jak liczymy złożoność: czy używamy struktury słownikowych, ale nie liczymy tego kosztu, czy raczej liczymy, czy używamy bardziej skomplikowanej struktury i koszty są liniowe.]]

Grafy zaimplementujemy w postaci modułu o następującej sygnaturze:

```
module type GRAPH =
  sig
    (* Typ grafów o wierzchołkach typu 'a. *)
    type 'a graph

    (* Pusty graf. *)
    val empty : 'a graph

    (* Dodanie wierzchołka (jeśli istnieje, to b.z.). *)
    val insert_vertex : 'a graph -> 'a -> 'a graph

    (* Dodanie krawędzi łączącej dwa (istniejące) wierzchołki. *)
    val insert_edge : 'a graph -> 'a -> 'a -> 'a graph

    (* Lista wszystkich wierzchołków. *)
    val vertices : 'a graph -> 'a list

    (* Lista incydencji danego wierzchołka. *)
    val neighbours : 'a graph -> 'a -> 'a list
  end;;
```

Problem przeszukiwania grafu.

24.2 Implementacja grafów

Graf reprezentujemy jako:

- zbiór wierzchołków pamiętamy wprost,
- zbiór krawędzi pamiętamy jako mapę, która każdemu wierzchołkowi przyporządkowuje zbiór jego sąsiadów.

```
type 'a graph = { v :  'a finset ; e : ('a, 'a finset) map }
```

Operacje na grafach implementujemy następująco:

```
module Graph : GRAPH =
  struct
    open BST
    open BstMap

    (* Typ grafów o wierzchołkach typu 'a. *)
    type 'a graph = { v :  'a finset ; e : ('a, 'a finset) map }

    (* Pusty graf. *)
    let empty = { v = empty_finset; e = BstMap.empty }

    (* Dodanie wierzchołka (jeśli istnieje, to b.z.). *)
    let insert_vertex g x =
      if member g.v x then g
      else { v = insert g.v x; e = update g.e x empty_finset }

    (* Dodanie krawędzi łączącej dwa (istniejące) wierzchołki. *)
    let insert_edge g x y =
      assert (member g.v x);
      assert (member g.v y);
      assert (x <> y);
      let e1 = update g.e x (insert (apply g.e x) y)
      in
        let e2 = update e1 y (insert (apply e1 y) x)
        in { v = g.v; e = e2}


    (* Lista wszystkich wierzchołków. *)
    let vertices g = finset2list g.v

    (* Lista incydencji danego wierzchołka. *)
    let neighbours g x =
      finset2list (apply g.e x)

  end;;
```

[[Dorobić generyczne przeszukiwanie grafów, sparametryzowane rodzajem struktury danych używanej do magazynowania wierzchołków.]]

[[A co z etykietowaniem krawędzi? A może moduł Graph?]]

24.3 Algorytm przeszukiwania wszerz BFS

[[Dorobić kolorowanie spójnych składowych, albo wyznaczanie listy spójnych składowych.]] Idea algorytmu przeszukiwania grafów wszerz przypomina rozprzestrzenianie się ognia w trakcie wypalania traw na łąkach. Jeśli ogień dojdzie do jakiegoś miejsca, to roprzestrzenia się na sąsiednie miejsca, na których jest jeszcze niewypalona trawa. W ten sposób przyłożenie ognia w jednym punkcie, spala cały „spójny obszar" suchej trawy.

Jeśli mówimy o grafach, to odwiedzenie jednego wierzchołka pociąga za sobą odwiedzenie jako kolejnych, jego (nie odwiedzonych jeszcze) sąsiadów. Istotna jest przy tym kolejność odwiedzania: najpierw odwiedzany jest wierzchołek początkowy, potem jego sąsiedzi, potem ich sąsiedzi itd.

Pamiętamy zbiór odwiedzonych wierzchołków. Dzięki temu, każdy wierzchołek jest odwiedzany co najwyżej raz. Dodatkowo wykorzystujemy kolejkę FIFO do przechowywania wierzchołków, które oczekują na odwiedzenie, ale nie zostały jeszcze odwiedzone.

Oto implementacja takiego algorytmu:

```
let bfs visit g =
  let visited = ref empty_finset
  in
    let search x =
      let q = ref empty_queue
      in
        begin
          q:= put_last !q x;
          visited := insert !visited x;
          while not (is_empty_queue !q) do
            visit (first !q);
            List.iter
              (fun y ->
                if not (member !visited y) then begin
                  q := put_last !q y;
                  visited := insert !visited y;
                end)
              (neighbours g (first !q));
            q := remove_first !q
          done
        end
    in
       List.iter (fun x -> if not (member !visited x) then search x) (vertices
g);;
```

Zauważmy, że każdy wierzchołek (poza wierzchołkami dla których wywoływana jest procedura `search`) jest rozpatrywany tyle razy, ile wychodzi z niego krawędzi, ale tylko raz zostanie wstawiony do kolejki `q`. Tak więc rozmiar kolejki nie przekroczy liczby wierzchołków. Stąd, algorytm ten ma złożoność czasową $O(n+m)$, a pamięciową $O(n)$.

Pojedyncze wywołanie procedury `search` przechodzi jedną spójną składową. Tak więc liczba spójnych składowych jest taka, jak liczba wywołań procedury `search`. Dodatkowo, jeżeli w trakcie odwiedzania będziemy „kolorować" wierzchołki, a przy każdym wywołaniu procedury `search` będziemy zmieniać kolor, to każda spójna składowa zostanie pokolorowana innym kolorem.

Algorytm przeszukiwania wszerz ma jeszcze jedną ciekawą właściwość. Dla każdego wywołania `search` x, wierzchołki są odwiedzane zgodnie z rosnącymi odległościami od wierzchołka x. Można pokazać, że w każdej chwili, w kolejce `q` znajdują się albo wierzchołki położone w tej samej odległości l od wierzchołka x, albo wierzchołki położone w odległości l i $l+1$,

przy czym te w odległości l znajdują się na początku kolejki. Niewątpliwie jest tak na samym początku, gdy w kolejce jest tylko wierzchołek x. W każdym kroku pętli `while` wyjmowany jest z początku kolejki wierzchołek położony w odległości l od wierzchołka x, a na koniec kolejki wstawiani są ci jego sąsiedzi, którzy nie zostali jeszcze rozpatrzeni, a więc znajdują się w odległości $l+1$ od wierzchołka x.

24.4 Algorytm przeszukiwania w głąb DFS

Algorytm przeszukiwania w głąb jest algorytmem przeszukiwania z nawrotami. Procedurę przeszukującą wywołujemy dla wierzchołka, w którym rozpoczynamy przeszukiwanie. Następnie jest ona rekurencyjnie wywoływana dla jego sąsiadów, ich sąsiadów itd. Przy tym, jeżeli jest ona wywoływana dla danego wierzchołka po raz kolejny, to nic się nie dzieje.

```
let dfs visit g =
  let visited = ref empty_finset in
  let rec search x =
    if not (member !visited x) then begin
      (* wierzchołek jeszcze nie był odwiedzony. *)
      visit x;
      visited := insert !visited x;
      List.iter search (neighbours g x)
    end
  in
    List.iter search (vertices g);;
```

Procedura `search` jest wywoływana $n + 2m$ razy, przy czym głębokość rekurencji nie przekracza $n+1$. Tak więc złożoność czasowa tego algorytmu jest rzędu $O(n+m)$, a pamięciowa rzędu $O(n)$.

Algorytm ten ma wiele ciekawych własności. Pozwala on na znalezienie w grafie cykli prostych, a także na podział grafu na dwuspójne składowe. Wyobraźmy sobie drzewa rekurencji procedury `search`. Są to drzewa rozpinające spójne składowe grafu. Zastanówmy się jak mogą być położone względem takiego drzewa pozostałe krawędzie w składowej. W ogólności, krawędź spoza drzewa rozpinającego może być położnona na dwa sposoby:

- może ona łączyć dwie różne gałęzie drzewa,
- może ona prowadzić w górę/dół drzewa.

[[Rysunek przedstawiający oba powyższe przypadki.]]

Czy krawędź w grafie może łączyć dwie różne gałęzie drzewa rozpinającego? Okzauje się, że nie. Gdyby tak było, to w trakcie obchodzenia pierwszej z tych gałęzi nastąpiłoby wywołanie procedury `search` dla wierzchołka leżącego na drugim końcu rozpatrywanej krawędzi. Tak więc stałaby się ona częścią drzewa rozpinającego wyznaczonego przez algorytm DFS.

Musi ona prowadzić w górę/dół drzewa rozpinającego. .....

[[Rysunek krawędzi rozpinającej cykl prosty.]]

[[DFS i sortowanie topologiczne.]]

[[Acykliczność]]

24.5 Algorytm Floyda-Warshalla

[[Uzupełnić]]

24.6 Algorytm Dijkstry

[[Uzupełnić]]

Ćwiczenia

1. [II OI, Klub Prawoskrętnych Kierowców, PCh] Mamy dany układ ulic, leżących na prostokątnej siatce $m \times n$. Ulice są jedno- lub dwukierunkowe. Przyjmujemy, że cztery strony świata są reprezentowane za pomocą liczb całkowitych od 0 do 3:

   ```
   let north = 0 and east = 1 and south = 2 and west = 3;;
   ```

   Układ ulic jest dany w formie trójwymiarowej tablicy wartości logicznych `ulice`: `ulice`.(i).(j).(k) określa, czy z punktu o współrzednych (i, j) można przejechać w kierunku k jedną jednostkę odległości. Można założyć, że tablica ta uniemożliwia wyjechanie poza obszar $\{0, \ldots, m-1\} \times \{0, \ldots, n-1\}$.

   Członkowie Klubu Prawoskrętnych Kierowców na skrzyżowaniach jeżdżą prosto lub skręcają w prawo. Sprawdź, czy z punktu $(0, 0)$ prawoskrętny kierowca może dojechać do punktu $(m-1, n-1)$.

2. [XIV OI, Grzbiety i doliny] Mamy daną mapę topograficzną terenu, w postaci prostokątnej tablicy liczb całkowitych, reprezentujących wysokości terenu. Grzbietem (doliną) nazwiemy taki obszar, który:

   - jest spójny (przyjmujemy, że kwadraty jednostkowe mapy sąsiadują ze sobą jeżeli stykają się bokami),
   - wszystkie kwadraty na tym obszarze mają tę samą wysokość,
   - jest on otoczony kwadratami o mniejszej (większej) wysokości.

   Wyznacz grzbiety i doliny na mapie.

3. [XIV OI, Biura] Firma ma n pracowników. Każdy pracownik ma telefon komórkowy, a w nim telefony pewnej grupy innych pracowników. Przy tym, jeżeli pracownik A ma telefon pracownika B, to pracownik B ma numer pracownika A. Firma chce podzielić pracowników na grupy, w taki sposób żeby:

   - każdych dwóch pracowników albo było w tej samej grupie, albo miało nawzajem swoje numery telefonów,
   - liczba grup była maksymalna.

   Wyznacz podział pracowników na grupy.

   Uwaga: Oczekiwana złożoność czasowa to $O(n^2)$, ale można to zadanie rozwiązać lepiej.

4. [II OI, Palindromy] Podziel dane słowo (listę znaków) na minimalną liczbę palindromów parzystej długości.

5. Dany jest acykliczny graf skierowany (DAG). Jego wierzchołki są ponumerowane od 0 do n, a krawędzie są dane w postaci tablicy sąsiedztwa (kwadratowej tablicy `e` wartości logicznych; w grafie mamy krawędź $u \to v$ wtw., gdy `e`.(u).(v)). Powiemy, że wierzchołek u *dominuje* wierzchołek v, jeżeli dla każdego wierzchołka w, z którego można dojść do v, z u można dojść do w.

   Napisz procedurę `zdominowani : bool array array -> int`, która na podstawie tablicy opisującej graf obliczy liczbę wierzchołków, dla których istnieją wierzchołki je dominujące.

6. [XIV OI, Powódź] Dana jest mapa topograficzna miasta położonego w kotlinie, w postaci prostokątnej tablicy typu `int * bool array`. Liczby określają wysokość kwadratów jednostkowych, a wartości logiczne określają, czy dany kwadrat należy do terenu miasta. Przyjmujemy, że teren poza mapą jest położony wyżej niż jakikolwiek kwadrat na mapie.

   Miasto zostało całkowicie zalane. Żeby je osuszyć należy w kotlinie rozmieścić pewną liczbę pomp. Każda z pomp wypompowuję wodę aż do osuszenia kwadratu jednostkowego, na którym się znajduje. Osuszenie dowolnego kwadratu pociąga za sobą obniżenie poziomu wody lub osuszenie kwadratów jednostkowych, z których woda jest w stanie spłynąć do kwadratu, na którym znajduje się pompa. Woda może przepływać między kwadratami, które stykają się bokami.

   Wyznacz minimalną liczbę pomp koniecznych do osuszenia miasta. Teren nie należący do miasta nie musi być osuszony. Miasto nie musi tworzyć spójnego obszaru.

7. [VIII OI, Spokojna komisja] ...

8. [IX OI, Komiwojażer Bajtazar] ...

9. [VIII OI, Mrówki i biedronka] ...

Wytyczne dla prowadzących ćwiczenia

Ad. 4 Spodziewamy się rozwiązania działającego w czasie $\Theta(n^2)$.

Wykład 25. Procedury dużo wyższych rzędów

Jaka jest różnica między danymi i procedurami? W programowaniu funkcyjnym to rozróżnienie rozmywa się. Dane mogą być traktowane jak procedury — każdy interpreter czy kompilator zamienia dane (kod źródłowy programu) na procedurę (wykonywalny program). Procedury wyższych rzędów operują na innych procedurach jak na danych. Tak więc procedury mogą być również danymi. Można by powiedzieć, że dane tym różnią się od procedur, że zawsze są podmiotem obliczeń, a nigdy nie są wykonywane. Niniejszy wykład powinien Państwa przekonać, że wcale tak nie musi być. Używając języka programowania nie widzimy w jaki sposób zrealizowane są struktury danych i na jak jest zrealizowane wykonywanie procedur. Łatwo sobie wyobrazić, że procedury mogą mieć postać kodu źródłowego lub częściowo skompilowanego, który jest *interpretowany*. Tak więc procedury mogą być zrealizowane w postaci danych (interpretowanych przez procedurę ewaluatora).

Podobnie, dane mogą być procedurami. Nie widzimy sposobu implementacji podstawowych struktur danych wbudowanych w język programowania. Poznamy teraz jedną z możliwych ich implementacji — całkowicie proceduralną.

Disclaimer: Niniejszy wykład ma charakter ćwiczenia umysłowego. Struktury danych wcale nie są implementowane w opisany sposób, ani nie jest to najlepszy sposób ich implementacji. Jest to jednak możliwe. Celem tego ćwiczenia jest zilustrowanie w pełni zamiennego traktowania danych jak procedur i procedur jak danych. Zdobywszy tę umiejętność będziemy mogli wznieść się na wyższy poziom abstrakcji i tworzyć struktury, w których dane są przemieszane z procedurami.

Niektóre programy, ze względu na system typów Ocamla działają „jednorazowo", tzn. każda konstrukcja jest poprawna i została przetestowana, jednak użycie jednej konstrukcji może uniemożliwić użycie innej.

25.1 Definiowanie form specjalnych

W Ocamlu argumenty procedur są obliczane *gorliwie*, tzn. najpierw są obliczane ich wartości, a dopiero potem przystępujemy do obliczania wartości procedury. Wyjątkiem są *formy specjalne*, np. `if` — w zależności od wartości warunku, tylko jeden z pozostałych członów jest obliczany. Czy można takie formy zaimplementować samemu? Tak. Potrzeba do tego dwóch rzeczy. Musimy umieć zabezpieczać wartości przed zbyt wczesnym obliczaniem i definiować *makrodefinicje*.

25.1.1 Makrodefinicje

W Ocamlu możemy zmieniać składnię języka wprowadzając własne makrodefinicje. Służy do tego preprocesor P4. Nie będziemy tu mówić o nim, lecz sporadycznie użyjemy definicji wprowadzających potrzebne nam formy specjalne.

25.2 Uleniwianie

Model obliczeniowy Ocamla charakteryzuje się zachłannym obliczaniem wartości — argumenty funkcji są obliczane bez względu na to, czy są potrzebne. Uleniwianie polega na zabezpieczeniu wartości przed obliczeniem, jeżeli nie jest to potrzebne. Zamiast wartości mamy „obietnicę jej obliczenia", czyli procedurę bezargumentową, której wartością jest dana wartość. W momencie

gdy wartość jest nam potrzebna, obliczamy ją. Żeby nie obliczać tej wartości wielokrotnie, stosujemy spamiętywanie. W Ocamlu dostępna jest forma specjalna `lazy` i procedura `force`. Lazy powoduje odroczenie (ze spamiętywaniem) obliczenia wyrażenia. Chcąc obliczyć wartość wykonujemy na wyniku `lazy` operację `force`. Prostsza wersja uleniwiania, bez spamiętywania, może wyglądać tak:

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

Pełne uleniwianie zawiera również spamiętywanie:

```
type 'a value = NoValue | Exception of exn | Value of 'a;;

let memoize f =
  let
    v = ref NoValue
  in
    function () ->
      match !v with
        Value y -> y |
        Exception e -> raise e |
        NoValue ->
          let
            y = try f ()
              with e -> begin
                v := Exception e;
                raise e
              end
          in
            v := Value y;
            y;;

lazy w ≡ memoize (fun () -> w)

let b = lazy (print_int 42; 42);;
force b;;
force b;;
```

25.3 Wartości logiczne

Aby mówić o wartościach logicznych potrzebujemy:

- prawdę i fałsz,
- `(if ...)`,

- koniunkcję, alternatywę i negację.

Wartość logiczną reprezentuję jako procedurę dwuargumentową, która w przypadku prawdy wybiera pierwszy, a w przypadku fałszu drugi argument.

```
let prawda x y = x;;
let falsz  x y = y;;
let jesli w k a = force (w k a);;
jesli prawda (lazy 1) (lazy 2);;
= 1
let i x y  = x y falsz;;
let lub x y = x prawda y;;
let nie x a b = x b a;;
jesli (lub falsz prawda) (lazy 1) (lazy 2);;
= 1
```

25.4 Produkty kartezjańskie

Aby mówić o produkcie musimy mieć:

- konstruktor pary `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 falsz;;
let p x = pr 1 "ala" x;;
p1 p;;
= 1
p2 p;;
= "ala"
```

Ze względu na system typów w Ocamlu, para `p` musi być zdefiniowana przy użyciu dodatkowego argumentu `x` (η-ekspansja).

25.5 Listy nieskończone

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

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

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

25.6 Liczby naturalne Churcha

Pozostały nam jeszcze liczby naturalne. (Rozszerzenie liczb naturalnych do całkowitych pozostawiamy jako ćwiczenie dla ambitnych. :-) Utożsamiamy liczbę naturalną z liczbą iteracji zadanej funkcji.

$$n \equiv f \to f^n$$

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

Dodawanie i mnożenie:

```
let inc n f = compose (n f) f;;
```
$f^{n+1} = f^n \circ f$

```
let plus m n f = compose (m f) (n f);;
```
$f^{m+n} = f^m \circ f^n$

```
let razy = compose;;
```
$$\begin{aligned}(\texttt{razy}\ m\ n)\ f &= (\texttt{compose}\ m\ n)\ f = \\ &= m\ (n\ f) = m\ (f^n) = \\ &= (f^n)^m = f^{m\cdot n}\end{aligned}$$

```
let potega m n = n m;;
```
$$\begin{aligned}(\texttt{potega}\ m\ n)\ f &= (n\ m)\ f = \\ &= \underbrace{(m \circ \cdots \circ m)}_{n \text{ razy}}\ f = \\ &= \underbrace{m(m \ldots (m}_{n \text{ razy}}\ f) \ldots) = \\ &= \underbrace{m(m \ldots (m}_{n-1 \text{ razy}}\ f^m) \ldots) = \\ &= \underbrace{m(m \ldots (m}_{n-2 \text{ razy}}\ f^{m^2}) \ldots) = \\ &\ \ \vdots \\ &= f^{m^n}\end{aligned}$$

Do celów testowych można używać:

```
let ile n = n (function x -> x + 1) 0;;
let compose f g = function x -> f (g x);;
let rec iterate n f =
  if n = 0 then id else compose (iterate (n-1) f) f;;
ile (iterate 1000);;
= 1 000

let dwa f = inc jeden f;;
ile (razy dwa dwa);;
- : int = 4
```

```
let trzy f = plus dwa jeden f;;
let szesc f = razy dwa trzy f;;
let siedem f = inc szesc f;;
ile (razy szesc siedem);;
- : int = 42

let osiem f = potega dwa trzy f;;
ile (plus dwa (razy (plus dwa trzy) osiem));;
- : int = 42
```

Jak moglibyśmy odróżniać od siebie liczby naturalne Churcha? Podstawową operacją porównania jest tzw. test zera — sprawdzenie, czy dana liczba jest równa zero. Szukamy więc takiej funkcji f, że $f^0 = \texttt{id}$ jest czymś istotnie innym, niż f^i dla $i > 0$. Taką funkcją może być:

$$f(x) = \texttt{falsz}$$

```
let test_zera x = x (function _ -> falsz) prawda;;
jesli (test_zera zero) (lazy 1) (lazy 2);;
= 1
```

Jak zmniejszyć n o 1? Liczba n to taki obiekt, który przekształca zadaną funkcję f w funkcję f^n. Nie mamy jednak żadnego dostępu do tego jak n to robi. Problem ten można przedstawić sobie tak: mając dane n, f i x należy obliczyć $f^{n-1}(x)$.

Rozważmy funkcję:

$$g(a, b) = (b, f\ b)$$

oraz ciąg:

$$(g^i(x, x))_{i=0,\ldots,n} = ((x, x), (x, f\ x), (f\ x, f^2\ x), \ldots, (f^{n-1}\ x, f^n\ x))$$

Wystarczy więc z ostatniej pary w ciągu wydobyć pierwszą współrzędną. Co można zapisać tak:

```
let dec n f x =
  let g (a, b) = (b, f b)
  in let (y, _) = n g (x, x)
     in y;;
val dec:(('a*'b->'b*'c)->'d*'d->'e*'f)->('b->'c)->'d->'e
```

lub tak, wykorzystując zaimplementowane przez nas produkty kartezjańskie:

```
let dec n f x = p1 (n (function p -> pr (p2 p) (f (p2 p))) (pr x x));;
val dec :
((((('a -> 'b -> 'b) -> 'c) -> ('c -> 'd -> 'e) -> 'e) ->
(('f -> 'f -> 'g) -> 'g) -> ('h -> 'i -> 'h) -> 'j) ->
('c -> 'd) -> 'f -> 'j = <fun>
ile (dec dwa);;
= 1
```

Odejmowanie, to wielokrotne zmniejszanie o jeden:

```
let minus m n = (n dec) m;;
ile (minus dwa jeden);;
= 1
```

Za pomocą odejmowania i testu zera można zaimplementować porównanie:

```
let wieksze m n = nie (test_zera (minus m n));;
jesli (wieksze dwa jeden) (lazy 1) (lazy 2);;
= 1
```

25.7 Abstrakcja rekurencji

Procedura `fold_left` stanowi abstrakcję rekurencyjnego przetwarzania list. Czy istnieje kwintesencja wszelkiej rekurencji? A co to jest rekurencja? Jak przerobić definicję rekurencyjną na nierekurencyjną — poprzez wprowadzenie dodatkowego parametru funkcyjnego. Przykład: silnia. Z rekurencją:

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

Bez rekurencji:

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

To jest procedura, która przetwarza procedurę `fac` liczącą poprawnie silnię dla liczb $\leq n$ na procedurę liczącą poprawnie silnię dla liczb $\leq n+1$. Teraz trzeba jakoś chytrze podstawić funkcję wynikową jako argument. Inaczej mówiąc szukamy funkcji, która jest punktem stałym `factorial`. Będzie ona poprawnie liczyć silnię dla wszystkich liczb.

Operator punktu stałego — pierwsze podejście.

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

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

To się niestety zapętli, ze względu na to, że argument procedury `factorial` jest obliczany przed jej zastosowaniem. Aby uniknąć zapętlenia, musimy uleniwić ten argument i wyliczać go tylko na żądanie. Operator punktu stałego.

```
let factorial fac n =
  if n <= 1 then
    1
  else
```

```
    n * (force fac (n - 1));;

let rec y f = f (lazy (y f));;

let fac = y factorial ;;
```

$$\texttt{fac}\ 3 = (\texttt{y factorial})\ 3 =$$
$$= (\texttt{factorial}(\texttt{lazy}(\texttt{y factorial})))\ 3 =$$
$$= 3 * (\texttt{force}(\texttt{lazy}(\texttt{y factorial}))\ 2) =$$
$$= 3 * (\texttt{y factorial}\ 2) =$$
$$= 3 * (\texttt{factorial}(\texttt{lazy}(\texttt{y factorial}))\ 2) =$$
$$= 3 * (2 * (\texttt{force}(\texttt{lazy}(\texttt{y factorial})))\ 1) =$$
$$= 3 * (2 * (\texttt{y factorial}\ 1)) =$$
$$= 3 * (2 * (\texttt{factorial}(\texttt{lazy}(\texttt{y factorial}))\ 1)) =$$
$$= 3 * (2 * 1) = 6$$

Każdą procedurę rekurencyjną możemy sprowadzić do zastosowania operatora punktu stałego i uleniwiania.

Oto kolejny przykład, rekurencyjna procedura obliczająca liczby Fibonacciego:

```
let fibonacci fib n =
  if n < 2 then
    n
  else
    (force fib (n-1)) + (force fib (n-2));;

let fib = y fibonacci;;
```

$$\texttt{fib}\ 3 = \texttt{y fibonacci}\ 3 =$$

$$= \texttt{fibonacci}(\texttt{lazy}(\texttt{y fibonacci}))\ 3 =$$

$$= \texttt{force}(\texttt{lazy}(\texttt{y fibonacci}))\ 2 + \texttt{force}(\texttt{lazy}(\texttt{y fibonacci}))\ 1 =$$

$$= \texttt{y fibonacci}\ 2 + \texttt{y fibonacci}\ 1 =$$

$$= \texttt{fibonacci}(\texttt{lazy}(\texttt{y fibonacci}))\ 2 + \texttt{fibonacci}(\texttt{lazy}(\texttt{y fibonacci}))\ 1 =$$

$$= \texttt{force}(\texttt{lazy}(\texttt{y fibonacci}))\ 1 + \texttt{force}(\texttt{lazy}(\texttt{y fibonacci}))\ 0 + 1 =$$

$$= \texttt{y fibonacci}\ 1 + \texttt{y fibonacci}\ 0 + 1 =$$

$$= \texttt{fibonacci}(\texttt{lazy}(\texttt{y fibonacci}))\ 1 + \texttt{fibonacci}(\texttt{lazy}(\texttt{y fibonacci}))\ 0 + 1 =$$

$$= 1 + 0 + 1 = 2$$

Ćwiczenia

- Zaimplementuj pary liczb całkowitych za pomocą operacji arytmetycznych i liczb całkowitych.
- Dla dowolnych dwóch funkcji $f : X \to A$ i $g : X \to B$ istnieje dokładnie jedna taka funkcja $h : X \to A \times B$, że $\pi_1 \circ h = f$ i $\pi_2 \circ h = g$. Zdefiniuj wprost procedurę `prod`, która na podstawie funkcji f i g wyznacza funkcję h dla proceduralnej definicji produktów. (`let prod f g x p = p (f x) (g x);;`)
- Zaimplementuj sumy rozłączne (koprodukty) za pomocą procedur. (Koprodukt zbiorów A i B to taki zbiór $A + B$ wraz z włożeniami $i_A : A \to A + B$, $i_B : B \to A + B$, że dla dowolnej pary funkcji $f : A \to X$ i $g : B \to X$ istnieje dokładnie jedna taka funkcja $h : A + B \to X$, że $h \circ i_A = f$ i $h \circ i_B = g$. Potraktuj tę definicję dosłownie.) Należy zaimplementować włożenie A i B w $A + B$ oraz procedurę, która na podstawie funkcji f i g wyznaczy funkcję h.

  Na potrzeby tego zadania możesz przyjąć, że A i B to ten sam typ.
- Potęgowanie funkcji w czasie logarytmicznym. Co tak na prawdę jest obliczane szybciej: funkcja wynikowa, czy jej wartość?
- Jak można rozszerzyć liczby naturalne Churcha do liczb całkowitych?
- Jak za pomocą operatora punktu stałego można wyznaczać procedury wzajemnie rekurencyjne?

Wykład 26. Strumienie

[[[Scalić z materiałami z Ważniaka.]]]

26.1 Czas rzeczywisty, a czas symulowany

Programowanie imperatywne i analogia obiektowa. Czas w modelowanym systemie jest modelowany przez czas w modelu systemu.

Programowanie strumieniowe — nie ma takiej analogii z czasem. Staramy się w modelu oddać strukturę zależności, a niekoniecznie kolejność zdarzeń.

Strumień, to ciąg wartości — cała, być może nieskończona, historia obiektu. Zamiast mówić o akcjach i zmianach stanu obiektów w danych chwilach patrzymy „jednym rzutem oka" na całą historię obiektu w czasoprzestrzeni. Staramy się oddać zależności między liniami świata jednych obiektów i innych.

26.2 Implementacja strumieni

Formalnie strumień to ciąg — być może nieskończony. Pomysł polega jednak na tym, żeby był on „leniwy", tzn. obliczane były tylko potrzebne wartości. Tak więc strumień to para: wartość i odroczony strumień pozostałych wartości.

```
type 'a stream = Nil | Cons of 'a * 'a stream Lazy.t;;

let empty s = s = Nil;;

let head s =
  match s with
    Nil -> failwith "Empty" |
    Cons (h, _) -> h;;

let tail s =
  match s with
    Nil -> failwith "Empty" |
    Cons (_, t) -> force t;;

EXTEND ... Makro Cons
```

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

```
let rec const_stream c =
  Cons (c, const_stream c);;

let ones = const_stream 1;;

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

```
    let h = head s
    and t = tail s
    in
      if p h then
        Cons (h, filter_stream p t)
      else
        filter_stream p t;;

let rec stream_ref s n =
  if n = 0 then
    head s
  else
    stream_ref (tail s) (n - 1);;

let rec stream_map f s =
  if empty s then
    Nil
  else
    Cons (f (head s), stream_map f (tail s));;

let rec stream_fold f a s =
  if empty s then
    Cons (a, Nil)
  else
    Cons (a, stream_fold f (f a (head s)) (tail s));;

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

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

26.3 Przykłady strumieni nieskończonych

Spróbujmy zdefiniować strumień liczb naturalnych:

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

let nats = integers_from 0;;
```

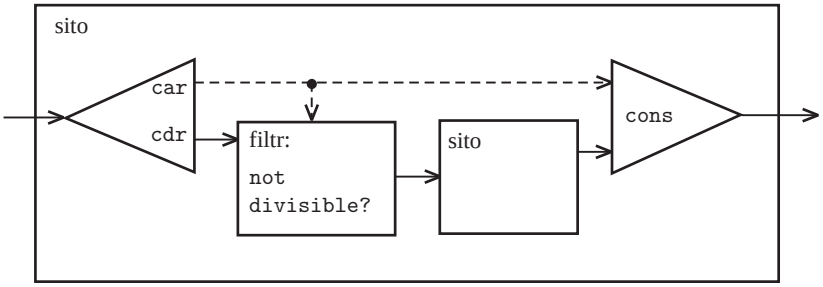
Podobnie możemy zdefiniować strumień liczb Fibonacciego:

```
let fibs =
  let rec fibo a b =
    Cons (a, fibo b (a+b))
  in fibo 0 1;;
```

Jak widać, jeżeli jesteśmy w stanie skonstruować iteracyjną procedurę wyliczającą kolejne elementy strumienia, to tym samym jesteśmy w stanie zdefiniować strumień tych wartości. Procedura taka stanowi ukrytą w strumieniu „maszynerię", która na żądanie podaje kolejne elementy strumienia. Takie definicje strumieni nazywamy *nie uwikłanymi* (w odróżnieniu od *uwikłanych*, które poznamy dalej). W przypadku definicji nie uwikłanych bytem rekurencyjnym jest procedura, a nie strumień. Strumień jest wtórny wobec procedury rekurencyjnej.

26.4 Strumień liczb pierwszych — sito Eratostenesa

Spróbujmy skonstruować nieskończony strumień liczb pierwszych, generowanych metodą sita Eratostenesa. Sposób konstrukcji takiego strumienia możemy przedstawić w postaci schematu przypominającego schematy blokowe układów przetwarzających sygnały. Strumienie są zaznaczone liniami ciągłymi, a pojedyncze wartości przerywanymi.



Schemat ten możemy zapisać w postaci następującego programu:

```
let divisible x y = x mod y = 0;;

let sito p s =
  filter_stream (function x -> not (divisible x p)) s;;

let rec sieve s  =
  Cons (head s, sieve (sito (head s) (tail s)));;

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

26.5 Definicje uwikłane

Definiując nieskończone strumienie nie musimy tego zawsze robić poprzez podanie odpowiedniej procedury rekurencyjnej (`integers-from`, `fibgen`, `sieve`). Możemy użyć elementów strumienia do zdefiniowania jego samego.

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

Do budowy bardziej skomplikowanych strumieni potrzebujemy dodatkowych operacji budujących strumienie. Na przykład, do utworzenia strumienia liczb naturalnych czy liczb Fibonacciego potrzebujemy dodawania strumieni.

```
let rec add_int_streams s1 s2 =
  if empty s1 or empty s2 then
    Nil
  else
    Cons (head s1 + head s2, add_int_streams (tail s1) (tail s2));;

let rec nats =
  Cons (1, add_int_streams ones nats);;
```

Zamiast dodawania strumienia jedynek, możemy użyć operacji zwiększania o jeden:

```
let succ x = x + 1;;
let rec nats =
  Cons(0, stream_map succ nats);;
```

A oto uwikłana definicja liczb Fibonacciego:

```
let rec fibs =
  Cons (0, Cons (1, add_int_streams fibs (tail fibs)));;
```

Możemy też w uwikłany sposób zdefiniować strumień liczb pierwszych. Użyjemy do tego predykatu sprawdzającego, czy liczba jest pierwsza:

```
let rec primes =
  Cons (2, filter_stream prime (integers_from 3))
```

Natomiast predykat `prime?` zdefiniujemy używając ... strumienia liczb pierwszych:

```
and prime n =
  let rec iter ps =
    if square (head ps) > n then
      true
    else if divisible n (head ps) then false
    else iter (tail ps)
  in
    iter primes;;
```

Całość działa poprawnie, ponieważ strumień jest leniwą strukturą danych. Pierwszy element, 2, jest dany explicite. Natomiast sprawdzając kolejne liczby, czy są pierwsze, zawsze mamy już obliczone potrzebne liczby pierwsze. Konkretnie, największa obliczona liczba pierwsza zawsze jest większa od pierwiastka z kolejnej liczby pierwszej, $2^2 = 4 > 3$, $3^2 = 9 > 5$, ...

26.6 Iteracje jako strumienie

W przypadku definicji nie uwikłanych używaliśmy procedur iteracyjnych do zdefiniowania strumieni. Możemy ten mechanizm odwrócić i użyć strumieni do opisania procesów iteracyjnych — kolejne elementy strumienia mogą reprezentować kolejne kroki iteracji. Oto przykład, strumień kolejnych przybliżeń pierwiastka, metodą Newtona:

```
let sqrt_improve g x = (g +. x /. g) /. 2.0;;

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

Spróbujmy przybliżyć liczbę π. Użyjemy do tego celu szeregu zbieżnego do $\frac{\pi}{4}$:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

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

let pi_summands =
  let
    succ x =
      if x > 0.0 then
        -.x -. 2.0
      else
        -.x +. 2.0
  in
    let rec s =
      Cons (1.0, stream_map succ s)
    in
      stream_map (fun x -> 1.0 /. x) s;;

let partial_sums s =
  let rec ps =
      Cons (head s, add_float_streams (tail s) ps)
  in ps;;

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

Taki strumień jest co prawda zbieżny, ale bardzo wolno. Po zsumowaniu pierwszych 1000 elementów ustalone są dopiero trzy pierwsze cyfry: 3.14. Jest to dosyć oczywiste, jeżeli zauważymy, że suma pierwszych n elementów jest obarczona błędem rzędu $\frac{1}{n}$

Można „przyspieszyć" zbieżność tego szeregu stosując „akcelerator Eulera". Jeśli S_n jest sumą pierwszych n wyrazów szeregu, to szereg zakcelerowany ma postać.

$$S_{n+1} - \frac{(S_{n+1} - S_n)^2}{S_{n-1} - 2S_n + S_{n+1}}$$

Działa on dobrze dla ciągów o malejących modułach błędów przybliżenia.

Dlaczego on działa? Przedstawmy kolejne sumy częściowe szeregu w postaci granica plus błąd: $S_n = x + e_n$, $|e_n| > |e_{n+1}| > 0$. Wówczas elementy przyspieszonego szeregu mają postać:

$$x + e_{n+1} - \frac{(x + e_{n+1} - x - e_n)^2}{x + e_{n-1} - 2x - 2e_n + x + e_{n+1}} =$$

$$= x + e_{n+1} - \frac{(e_{n+1} - e_n)^2}{e_{n-1} - 2e_n + e_{n+1}} =$$

$$= x + \frac{e_{n+1}e_{n-1} - 2e_{n+1}e_n + e_{n+1}^2 - e_{n+1}^2 + 2e_ne_{n+1} - e_n^2}{e_{n-1} - 2e_n + e_{n+1}} =$$

$$= x + \frac{e_{n+1}e_{n-1} - e_n^2}{e_{n-1} - 2e_n + e_{n+1}}$$

Jeżeli np. $|e_n| = e \cdot c^n$ i znaki e_n są takie same lub naprzemienne, to $e_{n+1}e_{n-1} - e_n^2 = 0$, czyli ciąg natychmiast osiąga granicę.

.... sprawdzić, kiedy można strumień wiele razy przyspieszać ...

Oto implementacja akceleratora Eulera:

```
let rec euler_transform s =
  let s0 = stream_ref s 0
  and s1 = stream_ref s 1
  and s2 = stream_ref s 2
  in Cons
    (s2 -. square (s2 -. s1) /. (s0 -. 2.0 *. s1 +. s2),
     euler_transform (tail s));;
```

Taki strumień jest już zbieżny w rozsądnym czasie. Przeliczenie 1000 elementów przyspieszonego szeregu daje przybliżenie 3.141592653. Jeszcze lepsze wyniki daje wielokrotne przyspieszanie. Skonstruujmy strumień kolejnych przyspieszeń strumienia sum częściowych i wybierzemy z niego strumień pierwszych elementów:

```
let rec pi_table =
  Cons (pi_stream, stream_map euler_transform pi_table);;

let pi_stream = stream_map head pi_table;;

let pi = stream_ref pi_stream 8;;
```

Taki strumień już w 8 elemencie osiąga granice precyzji arytmetycznej.

26.7 Strumienie par — liczby Ramanujana

Przedstawimy teraz jako przykład operowania strumieniami, konstrukcję strumienia liczb Ramanujana. Liczby Ramanujana to takie liczby naturalne, które można przedstawić jako sumy sześcianów dwóch liczb naturalnych na dwa różne sposoby. Pierwsze z tych liczb, to: 1729, 4104, 13832, 20683. Niech waga pary liczb naturalnych to będzie suma sześcianów jej składowych. Pomysł konstrukcji strumienia liczb Ramanujana polega na skonstruowaniu strumienia

(nieuporządkowanych) par liczb naturalnych uporządkowanego wg niemalejących wag, wychwyceniu par o powtarzających się wagach i wypisaniu tych wag.

Następująca procedura łączy dwa strumienie uporządkowane wg niemalejących wag elementów w jeden uporządkowany strumień. Waga jest określona w postaci funkcji `w`

```
let rec merge_weighted w s1 s2 =
  if empty s1 then s2 else
  if empty s2 then s1 else
    let h1 = head s1
    and h2 = head s2
    in
      if w h1 < w h2 then
        Cons (h1, merge_weighted w (tail s1) s2)
      else if w h1 > w h2 then
        Cons (h2, merge_weighted w s1 (tail s2))
      else
        Cons (h1, Cons (h2, merge_weighted w (tail s1) (tail s2)));;
```

Następujące definicje określają wagi par oraz funkcję tworzącą strumień par uporządkowanych wg wag.

```
let cube x = x * x * x;;

let weight (x, y) = cube x + cube y;;

let rec pairs s =
  Cons ((head s, head s),
        merge_weighted
          weight
          (stream_map (function x -> (head s, x)) (tail s))
          (pairs (tail s)));;
```

Następująca procedura pozostawia jedynie reprezentantów sekwencji elementów o powtarzających się wagach.

```
let rec non_uniq w s =
  let rec skip we s =
    if empty s then Nil
    else if we = w (head s) then skip we (tail s)
    else s
  in
    if empty s then Nil
    else if empty (tail s) then Nil
    else
      let h1 = head s
      and h2 = head (tail s)
      in
        if w h1 = w h2 then
          Cons (w h1, non_uniq w (skip (w h1) s))
```

```
        else
          non_uniq w (tail s);;
```

Strumień liczb Ramanujana możemy zdefiniować następująco:

```
let ramanujan =
  non_uniq weight (pairs nats);;
```

26.8 Co jeszcze ...

Strumienie jako reprezentacja danych, np. szeregów potęgowych. Operacje na funkcjach reprezentowanych w taki sposób.

Ćwiczenia

Zdefiniuj strumienie i narysuj schematy odpowiadające ich definicjom. (Tam, gdzie się da, w sposób uwikłany.)

- Strumień silni.
- Strumień postaci: jedna jedynka, dwie dwójki, trzy trójki itd. (Do rozwiązania uwikłanego potrzebne jest scalanie strumieni monotonicznych.)
- Napisz procedurę `codrugi: 'a stream -> 'a stream`, która przekształca strumień postaci $(s_1; s_2; s_3; s_4; \dots)$ w strumień postaci $(s_1; s_3; s_5; \dots)$. Powinna ona działać zarówno dla strumieni skończonych, jak i nieskończonych.
- Przeplot elementów dwóch strumieni (można sprytnie, zamieniając w wywołaniu rekurencyjnym miejscami argumenty).
- Strumień wszystkich par (uporządkowanych) elementów z dwóch danych strumieni (w dowolnej kolejności).
- Strumień liczb całkowitych, które w rozkładzie na liczby pierwsze mają tylko 2, 3 i 5 [R.Hamming].
- Dany jest nieskończony strumień liczb $s = (s_1, s_2, s_3, \dots)$. Jego strumień różnicowy, to strumień postaci: $s' = (s_2 - s_1, s_3 - s_2, s_4 - s_3, \dots)$. Strumień różnicowy drugiego rzędu, to strumień różnicowy strumienia różnicowego. Ogólniej, strumień różnicowy n-tego rzędu polega na n-krotnym wzięciu strumienia różnicowego, zaczynając od s. Zdefiniuj, w sposób uwikłany, strumień złożony z pierwszych elementów strumieni różnicowych kolejnych rzędów $(s_1, s_2 - s_1, (s_3 - s_2) - (s_2 - s_1), \dots)$. Narysuj diagram ilustrujący rozwiązanie.
- [IOI 2005] Uśrednienie strumienia $s = (s_1, s_2, \dots)$, to strumień (o jeden element krótszy) postaci: $\left(\frac{s_1+s_2}{2}, \frac{s_2+s_3}{2}, \dots\right)$.

  Dla danego strumienia liczb naturalnych s oblicz taki strumień $(x_1, x_2, \dots)$, że x_i jest liczbą takich niemalejących strumieni liczb całkowitych $(y_1, y_2, \dots, y_{i+1})$, których uśrednieniem jest $(s_1, s_2, \dots, s_i)$.
- Dany jest nieskończony strumień nieskończonych strumieni s. Zdefiniuj jego „przekątną” p.
- Szereg potęgowy $a_0 + a_1x + a_2x^2 + \dots$ możemy reprezentować jako strumień jego kolejnych współczynników; przy takiej implementacji szeregów potęgowych zaimplementuj:
  - pochodną,
  - całkę,
  - interpolacje wybranych funkcji (np.: e^x, $\ln x$, $\cos x$, $\sin x$),
  - mnożenie szeregów potęgowych,

– niech X będzie szeregiem potęgowym o pierwszym współczynniku równym 1; oblicz odwrotność X, tzn. taki szereg S, że $X \cdot S = 1$; niech $X = 1 + x \cdot X'$, wówczas:

$$(1 + x \cdot X') \cdot S = 1$$

$$S + x \cdot X' \cdot S = 1$$

$$S = 1 - x \cdot X' \cdot S$$

– korzystając z wyników poprzedniego zadania zaimplementuj dzielenie szeregów potęgowych.

- Na podstawie wzoru: $e = \sum_{i=0}^{\infty} \frac{1}{i!}$ skonstruować strumień kolejnych cyfr liczby e.

Wykład 27. Programowanie obiektowe

27.1 Ideologia programowania obiektowego

- Rozszerzenie paradygmatu programowania imperatywnego. Obiektom z modelowanego problemu odpowiadają obiekty obliczeniowe.
- Wyróżniamy klasy obiektów z modelowanego problemu. Między obiektami (lub ich klasami) zachodzą różnego rodzaju zależności. Typowe rodzaje zależności, to:
  - relacja zawierania,
  - relacja znajomości,
  - relacja dziedziczenia.
- Relacje zawierania i znajomości to relacje między pojedynczymi obiektami. Są one realizowane w ten sposób, że jeden obiekt obliczeniowy posiada dowiązanie do drugiego obiektu.
- Relacja dziedziczenia, to relacja między klasami obiektów.
- Metodologia projektowania i programowania obiektowego (w skrócie):
  - Wyróżnienie (klas) obiektów występujących w opisie problemu.
  - Projekt hierarchii klas i zależności między obiektami tworzącymi model danych.
  - Analiza przypadków użycia i typowych interakcji z systemem.
  - Dodanie obiektów sterujących i realizujących interfejs.
  - Analiza interakcji (komunikacji) między obiektami i metod potrzebnych do zrealizowania systemu.
  - Implementacja.

Przykład: Dokonać analizy hierarchii klas i zależności między obiektami dla gry będącej tematem programu zaliczeniowego.

27.2 Programowanie obiektowe

27.2.1 Klasy i obiekty

Obiekty (tej samej klasy) możemy traktować jak rekordy zawierające:

- pola z atrybutami,
- metody dostępu do obiektu.

Definicja klasy ma postać:

```
class punkt =
   object
     val mutable x = 0
     val mutable y = 0
```

```
    method polozenie = (x,y)
    method przesun (xv, yv) =
      begin
        x <- x + xv;
        y <- y + yv
      end
  end;;
```

Z obiektu korzystamy w następujący sposób:

```
let p = new punkt;;
p#przesun (1,0);;
p#przesun (3,2);;
p#polozenie;;
```

Treść definicji klasy jest wykonywana za każdym razem, gdy tworzony jest obiekt. Czasami chcemy, aby inicjacja obiektu (konstruktor) była sparametryzowana. Możemy to uzyskać parametryzując klasę. Przy tym parametry inicjacji są widoczne przez cały czas życia obiektu.

```
class punkt (x0, y0) =
  object
    val mutable x = x0
    val mutable y = y0
    method polozenie = (x,y)
    method przesun (xv, yv) =
      begin
        x <- x + xv;
        y <- y + yv
      end
    method na_poczatek =
      begin
        x <- x0;
        y <- y0
      end
  end;;

let p = new punkt (8,5);;
p#przesun (-4, -3);;
p#polozenie;;
p#na_poczatek;;
p#polozenie;;
```

27.2.2 Dziedziczenie

Istotnym elementem obiektowego paradygmatu programowania jest dziedziczenie. Dziedziczenie polega na tym, że definiując nową klasę możemy określić, że stanowi ona wzbogacenie istniejącej już klasy — dziedziczy jej cechy i metody. Oznacza to, że nowo definiowana klasa posiada wszystkie atrybuty i metody klasy, z której dziedziczy.

```
class kolorowy_punkt p c =
  object
    val color : int = c
    inherit punkt p
    method kolor = color
  end;;

let p = new kolorowy_punkt (3,1) 42;;
p#przesun (1,1);;
p#polozenie;;
p#kolor;;
```

Typ klasy dziedziczącej i tej, z której się dziedziczy są różne. W pewnym jednak sensie możemy traktować typ klasy dziedziczącej jak podtyp klasy, z której dziedziczy. Dlatego też mówimy o nadklasach i podklasach.

Możliwe jest również wielodziedziczenie — po prostu w treści definicji klasy może się pojawić wiele klauzul `inherit`.

Z pojęciem dziedziczenia związane jest pojęcie rzutowania typów. Typ podklasy można rzutować na typ nadklasy. Operator rzutowania typów ma postać:

```
(p : kolorowy_punkt :> punkt)
```

Jeżeli typ rzutowanego obiektu jest znany, to można go pominąć:

```
(p :> punkt)
```

27.2.3 Metody wirtualne

Metody wirtualne, to sposób tworzenia abstrakcyjnych nadklas. Metoda wirtualna ma określony typ, ale nie ma implementacji. Implementacja taka musi być zdefiniowana w podklasach. Dopóki wszystkie metody wirtualne nie zostaną zinstancjonowane, nie można tworzyć obiektów danej klasy.

```
class virtual punkt =
   object
     val mutable x = 0
     val mutable y = 0
     method polozenie = (x,y)
     method virtual przesun : int * int -> unit
   end;;

class p =
  object
    inherit punkt
    method przesun (xm, ym) =
      begin
        x <- x + xm;
        y <- y + ym
      end
  end;;
```

Literatura

[Apt81] Krzysztof R. Apt. Ten years of Hoare's logic: A survey — part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[BA05] Mordechai Ben-Ari. *Logika matematyczna w informatyce.* WNT, 2005.

[Bra83] J. M. Brady. *Informatyka teoretyczna w ujęciu programistycznym.* WNT, 1983.

[BS95] Holger Bickel and Werner Struckmann. The Hoare logic of data types. Technical Report 95-04, Technische Universität Braunschweig, Deutschland, 1995.

[CMP] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. Developing applications with objective caml. `http://caml.inria.fr/pub/docs/oreilly-book/`.

[CO81] Robert Cartwright and Derek Oppen. The logic of aliasing. *Acta Informatica*, 15:365–384, 1981.

[Dah92] Ole-Johan Dahl. *Verifiable Programming.* Prentice Hall, 1992.

[Dij85] Edsger W. Dijkstra. *Umiejętność programowania.* Wydawnictwa Naukowo-Techniczne, 1985.

[HA02] G. J. Sussman H. Abelson. *Struktura i interpretacja programów komputerowych.* WNT, 2002.

[Her65] H. Hermes. *Enumerability, Decidability and Computability.* Springer, 1965.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.

[Kub] Marcin Kubica. Programowanie funkcyjne. `http://wazniak.mimuw.edu.pl/index.php?title=Program`

[Kub00] Marcin Kubica. *Formalna specyfikacja wskaźnikowych struktur danych.* PhD thesis, Wydział Matematyki, Informatyki i Mechaniki, Uniwersytet Warszawski, 2000.

[Kub03] Marcin Kubica. Temporal approach to specification of pointer data-structures. In Mauro Pezzè, editor, *Fundamental Approaches to Software Engineering, FASE'2003*, number 2621 in LNCS, pages 231–245. Springer-Verlag, 2003.

[Ler] Xavier Leroy. The objective caml system. `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`

[MS87] Grażyna Mirkowska and Andrzej Salwicki. *Algorithmic Logic.* Państwowe Wydawnictwo Naukowe, 1987.

[MS92] Grażyna Mirkowska and Andrzej Salwicki. *Logika algorytmiczna dla programistów.* Wydawnictwa Naukowo-Techniczne, 1992.

[[Dodać CLR i ew. inne niewstawione skróty.]]

Ciekawe zadania:

- pręt i łuk,
- stacje benzynowe z różnymi cenami.

Wykład 28. Co jeszcze?

- Asercje w programach
- Pliki
- atrapy i strażnicy
- grafy, algorytmy DFS i BFS,
- obiegi drzew,
- konwersja wyrażeń z postaci infiksowej na prefiksową i postfiksową (ONP),
- problem plecakowy,
- optymalne mnożenie wielu macierzy.