Tajny Skrypt Lab SML[1]
v 0.33

Marcin Benke[2]
benke@mimuw.edu.pl
redakcja i rozdział 7

Grzegorz Grudziński[2]
gsg@mimuw.edu.pl
rozdział 8

Mikołaj Konarski[2]
mikon@mimuw.edu.pl
pozostałe rozdziały

10 października 2000

[1]Nazwa jest historyczna.
[2]Instytut Informatyki, Uniwersytet Warszawski.

Spis rzeczy

Ostrzeżenie

```
#include <std/disclaimer.h> ;-)
```

Ten dokument, zgodnie ze swym tytułem, nie tylko **nie jest oficjalnym skryptem**, ale co więcej — **jest tajny**.

Ponadto nie są to notatki do wykładu, tylko do **pracowni**. Wiedza podawana jest tu wyrywkowo i skrótowo, a nacisk położony jast na **zadania**.

Tajne Kolegium Redakcyjne

Rozdział 1

Wyrażenia

Program w języku funkcyjnym to wyrażenie, a wykonanie programu to obliczenie wartości wyrażenia. Przykłady wyrażeń:

```
3;   (* Comment: semicolon ";" tells the interpreter *)
(1 + 2) * 1;   (* to evaluate expressions. *)
("Hi! I'm the first element of this tuple", 3, false);
if ((2.3 > 5.4) orelse true) then "yes" else "no";
(fn n => 2 * n) 3;
fn x => x;
```

Wielkie i skomplikowane wyrażenia zapisuje się wygodniej używając konstrukcji `let...in...end` oraz deklaracji:

```
let
   val text = "I'm a string"
in
   text^". Yes, "^text^". Sure, "^text^"!"
end;
val twice = (fn n => 2 * n);
twice;
twice (twice 3);
fun twice' n = 2 * n;
twice 3 = twice' 3 andalso twice' 4 = twice 4;
```

Przykład z `pi` pokazuje pewną istotną ceche deklaracji. Jaka jest wartość wyrażenia `area 5.0` z ostatniej linii tego przykładu?

```
val pi = 3.14;
fun area r = pi * r * r;
area 5.0;
val pi = 1.0;
area 5.0;
```

Potężnym narzędziem programowania funkcjnego jest rekurencja:

```
fun fact n = if n <= 1 then 1 else n * (fact (n - 1));
fun fib n = if n <= 1 then 1 else (fib (n - 1)) + (fib (n - 2));
fact (fib 4);
fib;
```

Zadanie

Zadanie 1.1 Oto definicja funkcji `fib'`:

```
fun fib' n =
    let
        fun fib_pair n =
            if n <= 1 then (1, 1)
                      else let
                               val (j, k) = fib_pair (n - 1)
                           in
                               (j + k, j)
                           end
    in
        #1 (fib_pair n)
    end
```

(`#1` oznacza wzięcie pierwszego elementu tupla).

Udowodnij przy pomocy indukcji, że dla każdego n

$$\mathit{fib}\, n = \mathit{fib'}\, n.$$

Koszt czasowy $\mathit{fib}\, n$ jest wykładniczy względem n. Jaki jest koszt $\mathit{fib'}\, n$?

Rozdział 2

Listy & stuff

Poznaliśmy tuple:

```
(2, "errr", 3.14);
```

Listy to coś zupełnie innego:

```
[2, 55, 15346];
["This", "is", "a", "list", ", yeah."];
[];
```

Te nawiasy kwadratowe to skrótowy sposób zapisu list. Tak naprawdę listy konstruuje się w ten sposób:

```
2::(55::(15346::nil));
```

lub bez niepotrzebnych nawiasów:

```
2::55::15346::nil;
"This"::"is"::"a"::"list"::", yeah."::nil;
nil;
```

Mówiąc ściśle: infiksowy konstruktor `::` dostaje jako argumenty wartość x i listę l, której elementy są tego samego typu co x, i daje w wyniku l z doklejonym na początku elementem x. Na przykład (używając mieszanej notacji) `1::[2, 3, 4]` reprezentuje tę samą listę co `[1, 2, 3, 4]`

Kilka przykładów:

```
val mylist = [1, 2, 3];
fun null l = (l = []);
null mylist;
null [];
fun interval (n, m) = if n > m then [] else n::interval (n+1, m);
interval (~31, 72);
```

Jest troche wbudowanych funkcji na listach, w szczególności `hd` oddająca pierwszy element listy i `tl`, oddająca listę bez pierwszego elementu.

```
val head = hd mylist
val tail = tl mylist
val what = head :: tail
```

Funkcje na listach można definiowac za pomoca pattern matchingu

```
fun null [] = true
  | null (head::tail) = false
```

a tu będzie potrzebny wyjątek (exception):

```
exception Empty
fun hd [] = raise Empty
  | hd (head::tail) = head
```

Dalej już łatwo `:-)`

Zadania

Zadanie 2.1 Przy pomocy pattern matchingu napisz funkcję `tl`.

Zadanie 2.2 Używając funkcji `null`, `hd`, `tl`, napisz funkcję `sum`, która dodaje do kupy wszystkie elementy danej listy liczb całkowitych. Zrób to samo przy pomocy pattern matchingu.

Zadanie 2.3 Używając funkcji `null`, `hd`, `tl`, napisz funkcję `length`, która oblicza długość listy. Zrób to samo przy pomocy pattern matchingu.

Zadanie 2.4 Używając funkcji `null`, `hd`, `tl`, napisz funkcję `last`, która daje w wyniku ostatni element listy. Zrób to samo przy pomocy pattern matchingu.

Zadanie 2.5 Napisz funkcję `append`, która łączy dwie listy. Na przykład:

$$append\,([1,2],[3,4,6]) = [1,2,3,4,6]$$

Zadanie 2.6 Napisz funkcje `take` i `drop`, z parametrem `(l,n)`, które odpowiednio biorą lub odrzucają pierwsze `n` elementów listy `l`. Upewnij się, że dla każdego (l,n)

$$append\,(take\,(l,n), drop\,(l,n)) = l$$

Zadanie 2.7 Napisz, korzystając z funkcji `drop` i `hd`, funkcję `nth`, która daje n-ty element listy. Czy funkcja działałaby szybciej, gdyby skorzystać z `take` i `last`?

Zadanie 2.8 Napisz funkcję `flatten`, która dostaje listę list, a daje w wyniku listę wszystkich elementów tych list (nie martw się powtórzeniami czy kolejnością).

Zadanie 2.9 Napisz `quicksort`.

Zadanie 2.10 Przy pomocy list napisz funkcje realizujące arytmetykę na DUŻYCH liczbach całkowitych. Użyj ich do policzenia 500! albo lepiej 1000!. Jaka jest złożoność liczenia $n!$ w ten sposób?

Zadanie 2.11 Napisz to samo w C lub Pascalu. Porównaj czas zużyty na pisanie i wykonanie programu.

Poza tym można zrobić `mergesort` albo funkcje obliczające wszystkie permutacje, kombinacje, etc. danej listy albo parę funkcji: `zip` i `split`; `zip` dostaje dwie listy liczb całkowitych a oddaje listę par: para pierwszych elementów obu list, potem drugich, itd, `split` jest funkcją (prawie) odwrotną.

Wszystkie napisane programy warto przetestować — jest to łatwe i przyjemne, bo używamy interpretera.

Zachęcamy do wszelakich eksperymentów!

Rozdział 3

Rekursja ogonowa, akumulatory i różne takie

Często funkcję można zdefiniować na kilka istotnie rożnych sposobów. Sztandarowym przykładem jest funkcja dająca n–tą liczbę Fibonacciego, wśród której najprostszych definicji istnieją drastyczne rożnice złożoności czasowej. Dużą rodzinę innych przykładów stanowią funkcje przy których definiowaniu można ale nie trzeba używać *akumulatora* tzn. dodatkowego parametru który kumuluje składowe wyniku.

Oto definicja silni bez akumulatora:

```
fun fact n = if n <= 1 then 1 else n * fact (n - 1)
```

i z akumulatorem:

```
fun fact_tr n =
    let
        fun fact_acc (acc, k) = if k <= 0 then acc
                                else fact_acc (acc * k, k - 1)
    in
        fact_acc (1, n)
    end
```

przy czym widać, że dla każdego n mamy

$$\mathit{fact}\, n = \mathit{fact\_tr}\, n = n!.$$

Jak wygląda obliczanie `fact 4`? Funkcja `fact 4` wywołuje `fact 3`, zapamiętując, że wynik tego wywołania powinna pomnożyć przez 4, nim odda go jako swój wynik. Funkcja `fact 3` wywołuje `fact 2`, zapamiętując, że wynik tego wywołania powinna pomnożyć przez 3. Funkcja `fact 2` wywołuje `fact 1`, zapamiętując, że wynik tego wywołania powinna pomnożyć przez 2. Teraz `fact 1` oddaje 1. Następnie `fact 2` mnoży to przez 2 i oddaje rezultat tego mnożenia, czyli 2. Potem `fact 3` mnoży to przez 3 i daje 6. Potem `fact 4` mnoży to przez 4 i oddaje 24. A więc kolejność mnożeń była następująca: $((1 * 2) * 3) * 4$.

Twierdzę, że kolejność mnożeń przy obliczaniu `fact_tr 4` jest odwrotna, czyli $((4 * 3) * 2) * 1$. Zobaczmy: `fact_tr 4` wywołuje `fact_acc (1, 4)`, który po wykonaniu jednego mnożenia na rozruch wywołuje `fact_acc (4, 3)`. Następnie `fact_acc (4, 3)` mnoży 4 przez 3 i wywołuje `fact_acc (12, 2)`. Następnie `fact_acc (12, 2)` mnoży 12 przez 2 i wywołuje `fact_acc (24, 1)`. Następnie `fact_acc (24, 1)` mnoży 24 przez 1 i wywołuje `fact_acc (24, 0)`.

(*) A teraz `fact_acc (24, 0)` oddaje 24 i wynik ten zwracany jest już bez zmian przez kolejno `fact_acc (24, 1)`, `fact_acc (12, 2)`, `fact_acc (4, 3)`, `fact_acc (1, 4)` i `fact_tr 4`.

Skoro kolejność mnożeń jest odwrotna, `fact` i `fact_tr` rzeczywiście istotnie się od siebie różnią. Różnic jest więcej. Na przykład definicja `fact` jest niewątpliwie bardziej czytelna od `fact_tr`. A z drugiej strony w `fact_tr` występuje wyłącznie tzw. rekursja ogonowa (tail recursion), przez co `fact_tr` wykonuje się nieco szybciej i zużywa nieco mniej pamięci niż `fact`, w którym rekursja nie jest ogonowa.

Kiedy funkcja zawiera rekursję ogonową? Wtedy, kiedy w definicji funkcji wołanie rekurencyjne nie jest podwyrażeniem żadnego innego wyrażenia (z wyjątkiem drugiej lub trzeciej gałezi wyrażenia warunkowego). Jeśli rekursja jest ogonowa, wynik wykonania każdej funkcji wołanej jest przekazywany bez modyfikacji jako wynik funkcji wołającej, tak jak w miejscu oznaczonym (*).

Każdy przywoity kompilator zastępuje sekwencję powrotów taką jak (*) jedną instrukcją skoku, a także używa jednego rekordu aktywacji dla wszystkich rekurencyjnych wcieleń funkcji. W ten sposób rekurencja jest automatycznie zastępowana iteracją.

Zadania

Zadanie 3.1 Napisz wersję funkcji `length` z rekursją ogonową.

Zadanie 3.2 Napisz funkcję `decimal`, która dostaje listę cyfr i odaje liczbę jaka powstaje z tych cyfr, jeśli je ustawić za sobą (or something). np. `decimal [1, 9, 9, 5]` ma dać 1995.

Zadanie 3.3 Napisz dwie wersje funkcji `reverse` odwracającej listę. Jedna z tych wersji ma działać w czasie liniowym.

Zadanie 3.4 Czy można napisać ogonową wersję funkcji `take`? Jeśli tak, to czy warto jej używać? Odpowiedź uzasadnij pełnymi zdaniami.

Zadanie 3.5 Napisz wersję funkcji `interval` (z poprzedniego rozdziału) z rekursją ogonową. Nie używaj `reverse ;-)`.

Zadanie 3.6 Napisz funkcję `real_max`, która dostaje listę liczb rzeczywistych i oddaje największą z nich. Napisz dwie istotnie różne wersje. Która wersja bardziej Ci się podoba ?

Zadanie 3.7 To inna wersja funkcji `quicksort`:

```
fun quicksort [] = [] : int list
  | quicksort [x] = [x]
  | quicksort (a::rest) =
    let fun split(left,right,[]) =
            quicksort left @ (a::quicksort right)
          | split(left,right,x::l) =
            if x < a then split(x::left,right,l)
            else split(left,x::right,l)
    in split([],[],rest)
    end
```

Porównaj ją z wersją podaną w rozwiązaniu zadania z pracowni 2. Czytelność, ogonowość, akumulatorowość, etc.

Jak zwykle eksperymenty surowo wskazane!

Rozdział 4

Definiowanie typów

Można definiować typy:

```
datatype friend = PAT | SUSIE | JOE
datatype gift = BOTTLES_OF_BEER of int | BOOK
```

a przy pomocy pattern matchingu na tych typach tworzyć funkcje:

```
fun give PAT = BOTTLES_OF_BEER(10)
  | give SUSIE = BOOK
  | give JOE = BOTTLES_OF_BEER(3)
fun like (BOTTLES_OF_BEER(n)) = n > 7
  | like BOOK = true
```

Typy mogą być rekurencyjne:

```
datatype int_list = INT_NIL | INT_CONS of int * int_list
fun sum INT_NIL = 0
  | sum (INT_CONS(i, rest)) = i + sum rest
```

wzajemnie rekurencyjne:

```
datatype friend_rec = PAT | SUSIE | ANYONE_WHO_GIVES_ME of gift_rec
and gift_rec = BOTTLES_OF_BEER of int | PHOTO of friend_rec
fun love PAT = false
  | love SUSIE = true
  | love (ANYONE_WHO_GIVES_ME(a_gift)) = like a_gift
and like (BOTTLES_OF_BEER(n)) = n > 7
  | like (PHOTO(my_friend)) = love my_friend
```

polimorficzne :

```
datatype 'a option = SOME of 'a | NONE
val int_some_list = [SOME(5), NONE, SOME(10), SOME(3), NONE]
val bool_some_list = [SOME(true), SOME(true), NONE, SOME(true)]
fun divide (NONE, _) = NONE
  | divide (_, NONE) = NONE
  | divide (SOME(x), SOME(y)) = if y = 0.0 then NONE else SOME(x / y)
```

```
fun pair_some (SOME(x), SOME(y)) = SOME(x, y)
  | pair_some _ = NONE
fun divide2 (r, s) =
    case (pair_some (r, s))
      of NONE => NONE
       | (SOME(x, y)) => if y = 0.0 then NONE else SOME(x / y)
```

Znany nam wbudowany typ `list` jest rekurencyjny i polimorficzny i ma następującą definicję:

```
infixr 5 ::
datatype 'a list = nil | :: of ('a * 'a list)
```

A drzewo można zdefiniować na przykład tak:

```
datatype 'a tree = EMPTY | NODE of 'a tree * 'a * 'a tree
```

Zadania

Zadanie 4.1 Napisz wielkie mnóstwo funkcji na drzewach. Możesz napisać odpowiedniki wszystkich znanych Ci funkcji na listach. Możesz znaleźć jakieś funkcje specyficzne dla drzew, na przykład: obchodzące drzewo, zamieniające prawe poddrzewa z lewymi, etc. Możesz napisać funkcje z drzew w listy i z list w drzewa.

Zadanie 4.2 Zaimplementuj sortowanie drzewowe (treesort).

Wskazówka: W uporządkowanym drzewie binarnym liczba występująca w węźle jest większa od wszystkich w lewym poddrzewie i mniejsza od wszyskich w prawym. Należy zdefiniować dwie główne funkcje. Jedną, która dostaje listę liczb naturalnych, zaś oddaje uporządkowane drzewo binarne zawierające elementy tej listy. Drugą, która dostaje uporządkowane drzewo binarne, a oddaje posortowaną listę jego elementów. Treesort to złożenie tych funkcji.

Zadanie 4.3 To jest bardzo ładna (i odrobinkę nieefektywna) definicja liczb naturalnych:

```
datatype nat = ZERO | SUCC of nat
```

`SUCC` od successor - następnik. Napisz kilka podstawowych funkcji i relacji (czyli funkcji typu `nat->bool`) na tych liczbach naturalnych.

Zadanie 4.4 Oto drzewo składni abstrakcyjnej (gramatyka) pewnego języka wyrażeń

```
datatype expression =
    NUM of int
  | TIMES of expression * expression
  | PLUS of expression * expression
```

Napisz funkcję wyliczającą wartość wyrażenia. Napisz wersję $LL1$ powyższej gramatyki. Napisz funkcję przekształcającą wyrażenie w gramatyce $LL1$ w wyrażenie w podanej wyżej gramatyce. Napisz parser (używając gramatyki $LL1$) i lekser.

Wskazówka:

```
<1st_atom>   -> <numeral>
<2nd_atom>   -> <1st_atom> <2nd_atom'>
<2nd_atom'>  -> epsilon | * <1st_atom> <2nd_atom'>
<3rd_atom>   -> <2nd_atom> <3rd_atom'>
<3rd_atom'>  -> epsilon | + <2nd_atom> <3rd_atom'>
<expression> -> <3rd_atom>
```

Zadanie 4.5 Zdefiniuj typ drzew, w których każdy wierzchołek może mieć dowolnie wielu synów. Napisz trochę funkcji na tych drzewach.

Rozdział 5

Funkcje wyższych rzędów

Oto definicja funkcji `sum`:

```
fun sum [] = 0
  | sum (a::rest) = a + sum rest
```

Oto definicja funkcji `prod`:

```
fun prod [] = 1
  | prod (a::rest) = a * prod rest
```

Te funkcje są podejrzanie podobne.

Różnią się tylko nazwą i tym, że w deficji `sum` w pierwszym równaniu występuje `0` a w drugim `+`, podczas gdy w definicji `prod`, w tych samym miejscu w pierwszym równaniu jest `1` a w drugim `*`. Chciałoby się napisać funkcję `foldr` taką, żeby $sum = foldr\,(+, 0)$ i $prod = foldr\,(*, 1)$. Nic prostszego:

```
fun foldr (f, x) [] = x
  | foldr (f, x) (a::rest) = f(a, foldr (f, x) rest)
```

i tak powstała pierwsza funkcja wyższego rzędu `;-)`.

Teraz można napisać:

```
val sum = foldr (op +, 0)     (* op, because + is an infix identifier *)
val prod = foldr (op *, 1)
```

Zauważmy, że `foldr` tak jak go zapisaliśmy jest polimorficzny:

```
val sum_reals = foldr (op +, 0.0)
fun and_pair (b, c) = b andalso c
val and_list = foldr (and_pair, true)
val list_identity = foldr (op ::, nil)
val add_sizes = foldr (fn (s, rest_size) => (size s) + rest_size, 0)
```

(`fn x => E`, gdzie `E` to jakieś wyrażenie (być może zawierające `x`), oznacza funkcję która `x` przyporządkowuje `E`, jest to tak zwana anonimowa funkcja zapisana w tak zwanej lambda notacji).

Dla wygody przyjęto, że wyrażenie w rodzaju `(f a) b` (aplikacja funkcji `(f a)` do argumentu `b`) można zapisywać bez nawiasów: `f a b`. Przykład: `(foldr (op +, 0)) [2, 44, 5]` można zapisać jako: `foldr (op +, 0) [2, 44, 5]`.

Na zakończenie obrazowe podsumowanie działania `foldr`:

$$foldr\,(f,x)[a_1,a_2,a_3,a_4] = f(a_1,f(a_2,f(a_3,f(a_4,x))))$$

lub w nieco czytelniejszym, infiksowym zapisie:

$$foldr\,(+,x)[a_1,a_2,a_3,a_4] = a_1 + (a_2 + (a_3 + (a_4 + x)))$$

Zadania

Zadanie 5.1 Przy pomocy `foldr` napisz `append`.

Zadanie 5.2 Przy pomocy `foldr` napisz `length`.

Zadanie 5.3 Przy pomocy `foldr` napisz funkcję `good_max`, która dostawszy `n : int` oraz `l : int list` daje największą liczbę z listy `l` większą od `n`, lub `n` jeśli w `l` nie ma żadnej liczby większej od `n`.

Zadanie 5.4 Napisz funkcję `map`, która dostaje funkcję `f` i listę `l`, a daje listę wyników aplikacji funkcji `f` do kolejnych elementów `l`. Pisząc prosto:

$$map\ \ f\ [a_1,a_2,a_3,a_4] = [f\ a_1, f\ a_2, f\ a_3, f\ a_4].$$

Ten sam `map` zapisz przy pomocy `foldr`.

Zadanie 5.5 Przy pomocy `map` napisz funkcję `first_cut`, która dostaje listę list a daje listę pierwszych elementów tych list. Napisz `first_cut` nie używając `map`. Czy ta wersja nie przypomina Ci definicji `map`?

Zadanie 5.6 Napisz funkcję `filter`, która dostaje predykat `p : 'a -> bool` i listę `l : 'a list` i daje listę składającą się z tych elementów `l`, które spełniają `p`.

Napisz `filter` korzystając z `foldr`. Czy ta definicja jest bardziej czytelna?

Zadanie 5.7 Przy pomocy `filter` napisz `divis_list`, która wyławia z listy te liczby które są podzielne przez zadaną liczbę. Prawda, że ładnie?

Zadanie 5.8 Napisz `foldl`, czyli funkcję która zachowuje się w ten sposób

$$foldl\,(f,x)[a_1,a_2,a_3,a_4] = f(f(f(f(x,a_1),a_2),a_3),a_4)$$

lub w infiksowym zapisie:

$$foldl\,(+,x)[a_1,a_2,a_3,a_4] = ((((x + a_1) + a_2) + a_3) + a_4)$$

Zadanie 5.9 Używając między innymi `foldl` napisz `append`. Zwykły `append`.

Zadanie 5.10 Używając między innymi `foldl` napisz `foldr`. Korzystaj z doświadczeń z poprzedniego zadania.

Zadanie 5.11 Napisz odpowiedniki `foldr`, `map` i `filter` dla drzew.

Rozdział 6

Funkcje jeszcze wyższych rzędów

Funkcje wyższych rzędów spotyka się wszędzie. Weźmy choćby aplikację funkcji do argumentu. Toć to twór wyższego rzędu! Możemy go nazwać:

```
fun apply (f, a) = f a
```

i od tej pory zawsze pisać `apply (g, b)` tam gdzie dotąd pisaliśmy `g b  :-]`.

Operacja składania dwóch funkcji jest oczywiście funkcją wyższego rzędu. Pewna jej wersja jest nazwana i zdefiniowana w standardowej bazie SML-a:

```
infix 3 o
fun (f o g) x = f (g x)
```

i czasem się przydaje, np.:

```
val treesort = bst2list o list2bst
```

Nawet pospolite operacje arytmetyczne można zapisać jako funkcje wyższego rzędu:

```
fun divide_by x y = y / x;
divide_by 2.0 4.0;
val divide_by_two = divide_by 2.0;
divide_by_two 10000000.0;
divide_by_two 342423423452345.0;
map (divide_by 3.0) [333.0, 12.0, 55.0, 5.0];
```

Definiując `divide_by` na podstawie funkcji `/` dokonaliśmy pewnych myślowych operacji wyższego rzędu (jakżeby inaczej). Te operacje można zapisać w SML-u:

```
fun curry f a b = f (a, b)
fun C f x y = f y x
```

Definicję `divide_by` można teraz przedstawić tak:

```
val divide_by = C (curry (op /))
```

Tak na marginesie: dla dowolnego `f : 'a * 'b -> 'c` istnieje dokładnie jedna funkcja `g : 'a -> 'b -> 'c` taka, że dla każdego `(x, y) : 'a * 'b` `f (x, y) = g x y`. Tym `g` jest `curry f`. Skąd wynika na przykład, że każdą operację arytmetyczną można przerobić na funkcję wyższego rzędu tylko na dwa sposoby.

Jeszcze jedno — przy pomocy lambda notacji funkcje wyższych rzędów zapisuje się zagnieżdżając `fn`, np.:

```
fun curry f = fn a => fn b => f (a, b)
val C = fn f => fn x => fn y => f y x
```

Zadania

Zadanie 6.1 Zdefiniuj funkcje odwrotne do `curry` (będzie się nazywała `uncurry`) i `C` (nazwę wymyśl sama (sam)). Napisz funkcję `repeat`, która dostawszy pare `(f, n)` daje funkcję równą złożeniu `n` funkcji `f`. Czyli:

$$repeat\,(f, n) = f^n$$

Zadanie 6.2 Dla dowolnych dwóch typów `'a` i `'b` istnieje typ `'a * 'b`, ich produkt. W naturalny sposób związane są z tym typem pewne funkcje — projekcje:

```
fun pi1 (x, y) = x
fun pi2 (x, y) = y
```

i funkcja `pair` typu `('c -> 'a) * ('c -> 'b) -> ('c -> 'a * 'b)`, która dostawszy funkcje `f :'c -> 'a` oraz `g : 'c -> 'b` daje funkcję `h : 'c -> 'a * 'b`, o tej własności, że `pi1 o h = f` i `pi2 o h = g`. (Taka funkcja `h` jest dla każdego `(f, g)` tylko jedna.)

Napisz funkcję `pair`.

Zadanie 6.3 Zdefiniuj typ `('a, 'b) direct_sum`, który jest sumą rozłączną tyów `'a` i `'b`. Ten typ będzie w pewnym sensie dualny do typu `'a * 'b`. Zdefiniuj włożenia (injekcje):

```
inleft  : 'a -> ('a, 'b) direct_sum
inright : 'b -> ('a, 'b) direct_sum
```

i funkcję `cases` analogiczną (tak naprawdę dualną) do funkcji `pair` dla produktu.

Wskazówka 1: Funkcja `cases` jest typu
`('a -> 'c) * ('b -> 'c) -> ('a, 'b) direct_sum -> 'c`.

Wskazówka 2: Funkcja `cases` dostawszy funkcje `f : 'a -> 'c` oraz `g : 'b -> 'c` daje funkcję `h : ('a, 'b) direct_sum -> 'c` , o tej własności, że `h o inleft = f` i `h o inright = g` . (Taka funkcja `h` jest dla każdego `(f, g)` tylko jedna.)

Zadanie 6.4 Napisz funkcję

```
exists : ('a -> bool) -> 'a list -> bool
```

która bierze predykat `p : ('a -> bool)` i daje funkcję, która bierze listę `l` i oddaje `true` gdy któryś z elementów listy spełnia predykat `p`, `false` w przeciwnym przypadku. Napisz funkcję

```
non : ('a -> bool) -> ('a -> bool)
```

która neguje predykat. Napisz funcję `forall`, która tym się różni od `exists`, że w odpowiednim miejscu nie któryś, ale wszystkie elementy listy muszą spełniać predykat. Nim zaczniesz pisać `forall` podobnie jak `exists`, pomyśl czy nie możnaby go napisać śmieszniej (nie, nie chodzi mi o to żeby użyc `foldr`).

Spoiler: W definicji `forall` korzystaj wyłacznie z `exists`, `non`, i `o`.

Zadanie 6.5 Zapisz insertionsort lub inny algorytm sortowania (moźe mergesort, jeśli jeszcze go nie napisałeś lub nie napisałaś) w takiej postaci, aby jako pierwszy argument brał funkcję `le : 't * 't -> bool` wyznaczjącą pewien porządek i sortował listę `l : 't list` według tego porządku. Napisz `le_int` i `le_string`. Wypróbuj `insertionsort le_int`. Napisz też jakiś

```
le_int_to_int : (int -> int) * (int -> int) -> bool
```

Wypróbuj `insertionsort le_int_to_int`.

Rozdział 7

Operacje na tekstach

Operacje na tekstach w SML to w rzeczywistości operacje na listach znaków. Istnieje wbudowana funkcja `explode`, która dostając napis, daje w wyniku listę znaków, z których się on składa. Np. `explode "baba"` daje

```
[#"b",#"a",#"b",#"a"]
```

lub

```
["b","a","b","a"]
```

w zależnosci od implementacji.

Zadania

Zadanie 7.1 Używając funkcji `explode` zdefiniuj funkcję `size`, ktora oblicza długość napisu.

Zadanie 7.2 `implode` to funkcja odwrotna do `explode`. Napisz funkcję łączącą dwa napisy.

Zadanie 7.3 Napisz funkcję `linesplit : string -> string list`, przerabiającą napis na listę wierszy, z których się on składa.

Zadanie 7.4 Napisz funkcję `wordsplit : string -> string list`, przerabiającą napis na listę słów, z których się on składa.

Zadanie 7.5 Napisz funkcję

```
generic_split : string -> string -> string list,
```

przerabiającą napis na listę pól, z których się on składa, przyjmując drugi argument funkcji za zbiór separatorów pól.

Zadanie 7.6 Napisz funkcję sortującą leksykograficznie listę słów (można, a nawet należy skorzystać z napisanej wcześniej funkcji sortującej)

Zadanie 7.7 Napisz funkcję generującą dla danego napisu liste n najczęściej w nim występujących słów, wraz z liczbą wystąpień każdego z nich.

Zadanie 7.8 Napisz funkcję

```
strchr : string * char -> int list
```

taką, że `strchr(s,c)` daje listę pozycji, na których znak `c` występuje w napisie `s`.

Zadanie 7.9 Napisz funkcję

```
strstr : string * string -> int list
```

taką, że `strstr(s,p)` daje listę pozycji, na których zaczynają się wystąpienia napisu `p` w napisie `s`. Uwaga:

```
strstr("abababababa", "babab") = [2,4].
```

Rozdział 8

Leniwość

Rozdział 9

Moduły: struktury i sygnatury

Każdy duży a dobry program w naturalny sposób dzieli się na szereg odrębnych, w sporym stopniu niezależnych od siebie podprogramów. (Często podział ten następuje już w fazach projektowania i specyfikowania.) Każdy taki podprogram składa się z pewnej liczby typów, wartości, komentarzy, intuicji, itp. i wszystkie jego składniki są mocno ze sobą logicznie powiązane. Jeśli zamanifestujemy istnienie takiego podprogramu deklarując odpowiedni SML-owy moduł, to zarówno dla nas, jak i dla kompilatora stanie się lepiej widoczne, co dokładnie wchodzi w skład podprogramu, jakie są zależnosci między jego komponentami, itd. Jeśli dodatkowo przyjrzymy się krytycznie powstałemu (lub projektowanemu) modułowi i ocenimy co w nim jest istotne z punktu widzenia reszty naszego programu, a co jest jedynie szczegółem implementacyjnym, to będziemy mogli opatrzyć go względnie krótką, a mimo to *dobrą*, sygnaturą.

Przykład, który wiele wyjaśnia:

```
signature TOTAL_PREORDER =
sig

    type elem
    val leq : elem * elem -> bool
    (* and leq should be a total preorder *)

end;

signature PRIORITY_QUEUE =
sig

    structure Item : TOTAL_PREORDER

    type queue

    val empty : queue
    val insert : Item.elem * queue -> queue
    val join : queue * queue -> queue
```

```
    datatype min = MIN of Item.elem * queue | EMPTY
    val get_min : queue -> min

    (* By seq(q) we denote the sequence of Item.elem elements
       produced by iterating get_min on queue q.
       By num(t, q) we denote the number of occurrences of t in seq(q).

       For all q, p - queues constructed using the above operations,
       and for all t, u - elements of type Item.elem:
       1. seq(q) should be finite and non-decreasing,
       2. num(t, empty) should be 0,
       3. num(t, insert(u, q)) should be num(t, q), if u <> t,
          or num(t, q) + 1, if u = t,
       4. num(t, join(q, p)) should be num(t, q) + num(t, p). *)

end;

signature INTEGER_LEQ =
sig

    include TOTAL_PREORDER
    sharing type elem = int
    (* and leq should be the natural less-equal on int *)

end;

structure IntegerLeq : INTEGER_LEQ =
struct

    type elem = int
    fun leq (k, l) = ((k : int) <= l)

end;

signature INT_PRIORITY_QUEUE =
sig

    include PRIORITY_QUEUE
    sharing type Item.elem = int
    (* and Item.leq should be the natural less-equal on int *)

end;

structure IntPQasList : INT_PRIORITY_QUEUE =
struct

    structure Item = IntegerLeq (* type sharing and requirement *)
                                (* about Item.leq now satisfied *)

    type queue = Item.elem list (* queues will be unordered lists *)
```

```
    val empty = nil
    val insert = op ::
    val join = op @

    datatype min = MIN of Item.elem * queue | EMPTY
    fun get_min nil = EMPTY
      | get_min (k::tail) =
        case get_min tail
          of EMPTY => MIN(k, nil)
           | MIN(n, rest) => if Item.leq (k, n) then MIN(k, n::rest)
                             else MIN(n, k::rest)

end;
```

Nazwijmy naszkicowany powyżej program *IntPQasList*. Jest on poprawny tzn. dobrze implementuje kolejkę priorytetową liczb całkowitych. W szczególności `IntPQasList` pasuje do `INT_PRIORITY_QUEUE`, czyli zawiera wszystkie wymagane typy, wartości, itp., jak również spełnia wszystkie wymagania opisane w komentarzach (sprawdź).

Definicja (przez przykład) *dobroci*: Mówimy, ze sygnatura `INTEGER_LEQ` jest *dobra* dla struktury `IntegerLeq` w programie *IntPQasList*, gdyż:

1. `IntegerLeq` pasuje do `INTEGER_LEQ`
2. jesli napiszemy dowolną strukturę `A` pasującą do `INTEGER_LEQ`, to program *IntPQasList* z `IntegerLeq` podmienionym na `A` pozostanie poprawny

Dobra sygnatura pozwala nam i kompilatorowi rozumować o module, jako całości, wyłącznie na podstawie jego sygnatury, bez zaglądania do kodu. Wspomaga to abstrakcyjne myślenie o programie, bardzo ułatwia pielęgnację i modyfikację, pozwala na rozłączna kompilację modułów, etc.

Gdyby w programie *IntPQasList* zamiast `INTEGER_LEQ` zdefiniować następującą sygnaturę:

```
signature INTEGER_LEQ' =
sig

    include TOTAL_PREORDER
    sharing type elem = int
    (* and leq should additionally be antisymmetric *)

end;
```

to nie byłaby ona *dobrą* sygnaturą dla `IntegerLeq`. Oto dowód:

```
structure A : INTEGER_LEQ' =
struct

    type elem = int
    fun leq (k, l) = ((k : int) >= l) (* comment satisfied *)

end;
```

Widać, że `A` pasuje do `INTEGER_LEQ'`. Ale program *IntPQasList* z `INTEGER_LEQ'` zamiast `INTEGER_LEQ`, gdy podmienimy w nim `IntegerLeq` na `A`, przestaje być poprawny, gdyż porządek na `Item.elem` w module `IntPQasList` jest odwrotny niż wymagany w jego specyfikacji.

Pisz zawsze *dobre* sygnatury!

Zadania

Zadanie 9.1 Napisz moduły `COMPLEX_LEQ` (liczby zespolone uporządkowane według ich modułów), `COMPLEX_PRIORITY_QUEUE`, `ComplexLeq`, `ComplexPQasList`. Sprawdź poprawność i przetestuj.

Zadanie 9.2 Napisz program *IntPQasOrderedList*. Upewnij się, że moduł `IntPQasOrderedList` pasuje do sygnatury `INT_PRIORITY_QUEUE`. Pomyśl o złożoności czasowej *IntPQasList* oraz *IntPQasOrderedList*.

Zadanie 9.3 Napisz program sortujący liczby całkowite przy użyciu kolejki priorytetowej. Wykorzystaj *IntPQasList* jako jego podprogram. Czy `INT_PRIORITY_QUEUE` jest *dobrą* sygnaturą dla `IntPQasList` w tym programie?

Zadanie 9.4 Zmodyfikuj program z poprzedniego zadania tak, aby korzystał z *IntPQasOrderedList*. Jaki algorytm sortowania wynika z użycia `insert` a jaki z użycia `join` przy tworzeniu kolejki?

Rozdział 10

Moduły: funktory

Przy pomocy funktora można wyraźnie opisać, jakie moduły są potrzebne by zaimplementować dany moduł. Gdy program jest napisany w całości przy pomocy funktorów, w naturalny sposób przybiera on hierarchiczną postać. Jest jeden moduł, który rządzi i którego sygnatura jest opisem naszych oczekiwań co do programu, są moduły spełniające specyfikacje argumentów tego rządzącego funktora, są moduły spełniające specyfikacje argumentów tych modułów, itd. Dodatkowo, jeśli funktor ma dobrą sygnaturę i nie wymienia się w nim nazw żadnych zewnętrznych modułów (element dobrego stylu), to można o nim rozumować lub zmieniać jego implementację, abstrahując od własności czy istnienia jakichkolwiek innych modułów i nie psuje to poprawności programu.

Przykład funktora:

```
... (* <- TREE and Tree are defined here *)

functor PQasHeap
    ( structure TotalPreorder : TOTAL_PREORDER
      structure Tree : TREE ) :
sig

    include PRIORITY_QUEUE
    sharing Item = TotalPreorder

    (* queues should be constructed as trees  *)
    (* with value of every node less or equal *)
    (* to values of all the descendant nodes  *)

end =
struct

    structure Item = TotalPreorder (* sharing satisfied *)

    type queue = Item.elem Tree.tree
    ... (* still some work ;-) *)

end;
```

```
structure IntPQasHeap = PQasHeap (structure TotalPreorder = IntegerLeq
                                  structure Tree = Tree);
```

Zadania

Zadanie 10.1 Napisz program *PQasHeap* według szkicu podanego powyżej. Sprawdź, czy sygnatury są dobre. Przetestuj w przypadku liczb całkowitych i zespolonych.

Zadanie 10.2 Zmień `TREE` i `Tree` tak, by `Tree` był funktorem, o argumencie `Item`, a `tree` nie było polimorficznym typem danych, tylko typem zależnym od argumentu funktora:

```
datatype tree = EMPTY | NODE of tree * Item.item * tree.
```

Użyj tego `TREE` i `Tree` do zrobienia innej wersji programu *PQasHeap*. To jest niezbyt ładne bez funktorów wyższego rzędu, ale wykonalne.

Zadanie 10.3 Zmodyfikuj *PQasHeap* tak, by drzewa, jakimi są kolejki, były zawsze utrzymywane w postaci wyważonej. Postaraj się w jak największym fragmencie programu abstrahować od sposobu w jaki zapewniane jest wyważenie. Zamanifestuj tę abstrakcję odpowiednio deklarując moduły.

Zadanie 10.4 Napisz funktor implementujący sortowanie przy pomocy kolejek priorytetowych. Pomyśl o złożoności czasowej sortowania przy użyciu *PQasHeap*.

To już koniec :-[

Dodatek A

Jak zaliczyć laboratorium

Aby zaliczyć laboratorium SML należy napisać program. Powinien się on składać z kilku do kilkunastu małych modułów; dobrze wyspecyfikowanych, połączonych w sensowną całość jasnymi, wyraźnie opisanymi zależnościami.

Temat jest dowolny. Służymy radą i pomocą przy jego wyborze. Oto przykładowe (trochę nudne i wykorzystywane już zbyt wiele razy) zadanie zaliczeniowe:

Napisać *Kalkulator*, czyli program, który wśród swoich modułów ma moduł pasujący do sygnatury `CALCULATOR`, zdefiniowanej następująco:

```
signature CALCULATOR =
sig

    val eval : string -> string

    (* if s represents an arithmetic expression    *)
    (* (which means that s is a string of digits,  *)
    (* ~ (negation), +, -, * and parenthesis,      *)
    (* satisfying some well known properties),     *)
    (* then (eval s) should be equal to the string *)
    (* representing the value of s, or the string  *)
    (* "division by zero" if appropriate,          *)
    (* if s does not represent a valid arithmetic  *)
    (* expression, (eval s) should be equal to the *)
    (* string "syntax error"                       *)

end;
```

Wszelkie drobne niejasności tej specyfikacji należy interpretować na swoją korzyść. W razie problemów zwrócić się o pomoc.

Wydaje się, że wśród modułów składających się na *Kalkulator* powinny być moduły `Lexer`, `Parser`, `Evaluator` i `PrettyPrinter`. Byłoby ładnie i chyba wygodnie, gdyby `Parser` posługiwał się drzewem składni reprezentującym gramatykę $LL(1)$ wyrażeń, zaś `Evaluator` drzewem reprezentującym prostszą gramatykę:

```
datatype expression =
    NUM of int
  | NEGATION of expression
  | TIMES of expression * expression
  | PLUS of expression * expression
  | ...
```

To zadanie nie powninno być zbyt trudne. To co można tu zrobić lepiej lub gorzej to podział programu na moduły, zdefiniowanie zależności między modułami przy pomocy funktorów, opisanie własności modułów sygnaturami.

Powodzenia!

Dodatek B

Rozwiązania niektórych zadań

Wszystkie te rzeczy przeszły przez kompilator (w wypadku programów w SML to bardzo dobry omen), ale nie były zbyt intensywnie testowane. Dzięki za wskazywanie błędów!

```
(* Exercise 2.1 *)

exception Empty
fun tl [] = raise Empty
  | tl (head::tail) = tail

(* Exercise 2.2 *)

fun sum l = if null l then 0 else hd l + sum (tl l)

(* or *)

fun sum [] = 0
  | sum (i::rest) = i + sum rest

(* Exercise 2.3 *)

fun length l = if null l then 0 else 1 + length (tl l)

(* or *)

fun length [] = 0
  | length (head::tail) = 1 + length tail

(* Exercise 2.4 *)

fun last l = if null l then hd l else last (tl l)
```

```
(* or *)

fun last [] = raise Empty
  | last [elem] = elem
  | last (head::tail) = last tail

(* Exercise 2.5 *)

fun append ([], l) = l
  | append (head::tail, l) = head :: (append (tail, l))

(* Exercise 2.6 *)

fun take (l, 0) = nil
  | take (head::tail, n) = head :: take (tail, n-1)
  | take (nil, n) = nil

fun drop (l, 0) = l
  | drop (head::tail, n) = drop (tail, n-1)
  | drop (nil, n) = nil

(* Check, that the required equality          *)
(* holds even if n < 0. This is an ugly hack, *)
(* I know, but I hate exceptions even more:P. *)

(* Exercise 2.7 *)

fun nth (l, n) = hd (drop (l, n-1))

(* And this version is slower. Why? *)

fun nth2 (l, n) = last (take (l, n))

(* Exercise 2.8 *)

fun flatten [] = []
  | flatten (l::rest) = append (l, flatten rest)

(* Exercise 2.9 *)

(* @ is the built in, infix version of append *)

fun quicksort [] = [] : int list
  | quicksort [x] = [x]
  | quicksort (a::rest) =
    let
        fun split (b, []) = ([], [])
          | split (b, x::l) = let
                                  val (left, right) = split (b, l)
                              in
```

```
                                  if x < b then (x::left, right)
                                  else (left,x::right)
                              end
    in
        let
            val (left, right) = split (a, rest)
        in
            quicksort left @ (a::quicksort right)
        end
    end

(* Exercise 3.1 *)

fun length l = let
                   fun len_acc (len, []) = len
                     | len_acc (len, head::tail) = len_acc (len + 1, tail)
               in
                   len_acc (0, l)
               end

(* Exercise 3.2 *)

fun decimal l =
  let fun diggy (acc, []) = acc
        | diggy (acc, digit::rest) = diggy (10*acc + digit, rest)
  in
      diggy(0, l)
  end

(* Definitions of decimal without an accumulator are absolutely hopeless *)

(* Exercise 3.3 *)

(* reverse2 is O(n^2) *)

fun reverse2 [] = []
  | reverse2 (head::tail) = (reverse2 tail) @ [head]

(* Thanks to the use of accumulator reverse is O(n) *)
(* and by chance there is tail recursion here *)

fun reverse l = let
                    fun rev_acc (rev, []) = rev
                      | rev_acc (rev, head::tail) = rev_acc (head::rev, tail)
                in
                    rev_acc ([], l)
                end
```

```
(* Exercise 3.4 *)

(* This works. Notice reverse.               *)
(* A similar thing could be done for append. *)
(* Not worth the effort, is it?              *)

exception Empty

fun take (l, n) =
    let
        fun take_acc (acc, (l, 0)) = reverse acc
          | take_acc (acc, (hd::tl, n)) = take_acc (hd :: acc, (tl, n-1))
          | take_acc (acc, (nil, n)) = raise Empty
    in
        take_acc ([], (l, n))
    end

(* Exercise 3.5 *)

fun interval (from, to) =
    let
        fun inter_acc (l, (n, m)) = if n > m then l
                                    else inter_acc (m::l, (n, m-1))
    in
        inter_acc ([], (from, to))
    end

(* Exercise 3.6 *)

val minus_infinity = ~1000000000000000000000000000000.0

fun real_max [] = minus_infinity
  | real_max (head::tail) =
    let fun maxx (r, []) = r
          | maxx (r, x::tl) = if r > x then maxx (r, tl)
                              else maxx (x, tl)
    in
        maxx (head, tail)
    end

fun real_max [] = minus_infinity
  | real_max (x::tail) = let
                        val y = real_max tail
                    in
                        if y > x then y
                        else x
                    end
```

```
(* Exercise 5.1 *)

fun foldr (f, x) [] = x
  | foldr (f, x) (a::rest) = f(a, foldr (f, x) rest)

fun append (l1, l2) = foldr (op ::, l2) l1

(* Exercise 5.2 *)

val length = foldr (fn (elem, rest_length) => 1 + rest_length, 0)

(* Exercise 5.3 *)

fun good_max n = foldr (max, n)

(* Exercise 5.4 *)

fun map f nil = nil
  | map f (head::tail) = (f head)::(map f tail)

fun map f = foldr (fn (a, result) => (f a):: result, nil)

(* Exercise 5.5 *)

val first_cut = map hd

fun first_cut nil = nil
  | first_cut (head::tail) = (hd head)::(first_cut tail)

(* Exercise 5.6 *)

fun filter p nil = nil
  | filter p (a::rest) = if p a then a::(filter p rest)
                         else (filter p rest)

fun filter p = foldr (fn (a, filtered_tail) =>
                      if p a then a::filtered_tail
                      else filtered_tail, nil)
(* No, this didn't turn out more readable (at least for me). *)

(* Exercise 5.7 *)

fun divisible_by k l = l mod k = 0
fun divis_list n = filter (divisible_by n)
(* Yeees. *)

(* Exercise 5.8 *)

fun foldl (f, x) nil = x
  | foldl (f, x) (head::tail) = foldl (f, (f (x, head))) tail
```

```
(* Exercise 5.9 *)

fun reversed_cons (tail, head) = head :: tail
fun append (l1, l2) = foldl (reversed_cons, l2) (rev l1)

(* Exercise 6.1 *)

fun uncurry f = fn (a, b) => f a b

fun id x = x
fun repeat (f, 0) = id
  | repeat (f, n) = f o (repeat (f, n - 1))

(* Exercise 6.2 *)

fun pair (f, g) x = (f x, g x)

(* Exercise 6.3 *)

datatype ('a, 'b) direct_sum = INL of 'a | INR of 'b
val inleft = INL
val inright = INR
fun cases (f, g) (INL(x)) = f x
  | cases (f, g) (INR(y)) = g y

(* Exercise 6.4 *)

fun exists p [] = false
  | exists p (a::rest) = if p a then true
                         else exists p rest
fun non p = not o p
val forall = non o exists o non
```

Dodatek C

Jak pracować z kompilatorem SML-a

- Interaktywnie:
  1. uruchomić SML (`mosml`, `new-sml`, `sml-0.93`, `sml`)
  2. po zachęcie, którą na ogół jest myślnik, napisać program
  3. zakończyć pisanie średnikiem i nacisnąć `RET`
- Wsadowo:
  1. napisać program jakimś edytorem
  2. zapisać go w pliku `smth.sml`
  3. uruchomić SML i napisac `use "smth.sml";`
- Przy pomocy EMACS-a i `sml-mode`:
  1. uruchomić EMACS-a
  2. `C-x C-f smth.sml RET` (ważne, żeby końcówką nazwy było `.sml`)
  3. napisać błędny program (`Tab` automagicznie formatuje)
  4. `C-c C-l` i będzie wiadomo co dalej
  5. Zrobić `` C-c ` `` (ten ostatni znaczek to back-quote, występuje na klawiaturze na ogół pod tyldą), wtedy on ustawi kursor tam gdzie w pliku jest błąd
  6. Korzystać z innych dobrodziejstw `sml-mode`, np. `sml-mode-info`, czyli podręcznika do `sml-mode`