Zadanie domowe 1

Podczas dzisiejszych zajęć zaczniemy się zaznajamiać z programowaniem funkcyjnym w Standard ML poprzez implementowanie funkcji o rozmaitych typach. Każda z funkcji, jaką musicie się zająć, daje się zaimplementować za pomocą dosłownie kilku linijek kodu, tak więc dzisiejsze zadanie nie powinno polegać na wklepywaniu tysięcy linii kodu, ale stać się dobrą zabawą opartą przede wszystkim o staranne przemyślenie tego, co chce się napisać.

Zanim zabierzemy się do zadań, musimy przygotować sobie warsztat pracy:

- Ściągnijcie pliki z archiwum

  `www.math.us.edu.pl/~pgladki/teaching/2013-2014/log_lab1.zip`

  i je przeczytajcie. To zadanie składa się z dwóch części, pliki powinny się więc znaleźć w dwóch katalogach: *part_one* oraz *part_two*.
- Zainstalujcie Standard ML of New Jersey na komputerze. Generalnie będziemy używać wersji 110.67:

  `http://smlnj.org/dist/working/110.67/index.html`
- Chcąc kompilować Wasz kod, uruchomcie SML-a w tym samym podkatalogu, do którego rozpakowaliście pliki z `log_lab1.zip`, a więc w *part_one* dla pierwszej części zadania oraz *part_two* dla drugiej części zadania.
- Wpiszcie **CM.make "sources.cm";** To polecenie powinno wczytać i skompilować wszystkie pliki źródłowe dla danego katalogu.

Jako że Wasz wykładowca jest osobą z natury rzeczy leniwą, Wasze zadania programistyczne będą oceniały roboty – dlatego jest szczególnie ważne, abyście dokładnie przestrzegali zasad, według których powinniście składać Wasze zadania. Jeżeli nie dopilnujecie wszystkich kroków potrzebnych do prawidłowego złożenia zadania, robot zgłupieje i w rezultacie możecie otrzymać 0 punktów za zadanie.

Po ściągnięciu i rozpakowaniu archiwum `log_lab1.zip` powinniście znaleźć pliki `czesc1.sml`, `czesc1-sig.sml` oraz `sources.cm` w katalogu `part_one` oraz `czesc2.sml`, `czesc2-sig.sml` oraz `sources.cm` w katalogu `part_two`. Będziecie pisali Wasz kod w plikach `czesc1.sml` i `czesc2.sml`, nie zmieniajcie niczego w plikach `czesc1-sig.sml`, `czesc2-sig.sml` ani w plikach `sources.cm`. Szkielet pliku `czesc1.sml` wygląda mniej więcej tak:

```
structure foo :>PART_ONE =
struct
  exception NotImplemented
  datatype 'a tree= Leaf of 'a | Node of 'a tree * 'a * 'a tree
  fun sum _ = raise NotImplemented
  fun fac _ = raise NotImplemented
  ...
end
```

a szkielet pliku `czesc2.sml` tak:

```
structure foo :>PART_TWO =
struct
  exception NotImplemented
  datatype 'a tree = Leaf of 'a | Node of 'a tree * 'a * 'a tree
```

```
  funfact =raiseNotImplemented
  ...
end
```

W każdym z plików `czesc1.sml` i `czesc2.sml` zamieńcie `foo` w nazwie struktury na Wasz numer indeksu poprzedzony słowem kluczowym `id`. Przykładowo, jeżeli numer Waszego indeksu to 12345, zmodyfikujcie pierwszą linię pliku `czesc1.sml` następująco:

```
structure id12345 :>PART_ONE =
```

Jest to bardzo ważne! Jeżeli zapomnicie o tym kroku, robot oceniający zadanie nie będzie w stanie poprawnie zidentyfikować Waszego zadania i otrzymacie za nie 0 punktów.

Następnie wypełnijcie zawartość funkcji Waszym kodem, ale tylko wtedy, gdy udało Wam się poprawnie zaimplementować daną funkcję. **Jest to jeszcze ważniejsze!** Jeśli wyślecie kod, który się nie kompiluje, robot oceniający zgłupieje i dostaniecie za całe zadanie 0 punktów. Jeśli nie potraficie zaimplementować danej funkcji, po prostu zostawcie ją w niezmienionym kształcie.

Zadanie prawdopodobnie zajmie Wam ładnych kilka godzin, tak więc nie zwlekajcie z nim do ostatniej chwili. Starajcie się zaimplementować możliwie najwięcej funkcji. Celem uruchomienia programów w interpreterze Standard ML użyjcie Compile Manager. Za każdym razem, gdy kompilujecie Wasz kod źródłowy, powinniście zastosować `open` do Waszej `structure`.

Oto przykładowa sesja z wykorzystaniem `sources.cm` w katalogu `part_one`:

```
[foo:38 ] sml
Standard ML of New Jersey v110.58 [built: Fri Mar 03 15:32:15 2006]
- CM.make "sources.cm";
[autoloading]
.....
[New bindings added.]
val it = true : bool
- open foo;
opening foo
  exception NotImplemented
  datatype 'a tree = Leaf of 'a | Node of 'a tree * 'a * 'a tree
  val sum : int ->int
  val fac : int ->int
  ...
- sum 10;
val it = 55 : int;
- fac 10;
uncaught exception NotImplemented
  raised at: hw1.sml:5.27-5.41
```

Gdy uznacie, że zrobiliście już wszystko, co potrafiliście zrobić, wyślijcie pliki `czesc1.sml` i `czesc2.sml` emailem do prowadzącego. Termin składania zadania mija **1 stycznia 2014 roku**. Proszę pamiętać, aby w polu **subject** Waszego maila umieścić tag `[aghlog]`. **Pod żadnym pozorem** nie kompresujcie wysyłanych plików, nie zmieniajcie ich nazw, nie wysyłajcie całego katalogu `lab1` ani nie róbcie żadnej

z nieskończonego ciągu nieprzewidywalnych rzeczy, które moglibyście zrobić – po prostu wyślijcie maila z dołączonymi dwoma odpowiednio zmodyfikowanymi plikami i to wszystko.

1. Część pierwsza

Zanim zaczniecie, przeczytajcie starannie notatki z Wykładu 1 i Wykładu 2, a następnie Chapter 1 z podręcznika Benjamin C. Pierce, *Types and Programming Languages,* MIT Press, 2002 oraz Chapter 1, Chapter 2, Section 3.1i Section 3.2 z podręcznika Riccardo Puccella, *Notes on Programming SML/NJ*:

`http://www.cs.cornell.edu/riccardo/smlnj.html`

Wasz kod powinien być napisany zgodnie z wytycznymi stylu Standard ML. Nie jest to najważniejsze wymaganie, ale warto pamiętać, że dobry styl programistyczny pomaga nie tylko czytelnikowi zrozumieć kod, ale także Wam poprawić funkcjonalność kodu.

Dla tej części zadania **nie wolno** używać Wam żadnych z funkcji dostarczanych przez bibliotekę Standard ML.

Po tym, nieco przydługim, wstępie zabieramy się do zabawy:

1.1. Funkcje liczb całkowitych.

1.1.1. *Suma liczb naturalnych od* 1 *do* n. (2 punkty)
(Typ) `sum : int ->int`
(Opis) `sum` n zwraca $\sum_{i=1}^{n} i$
(Niezmiennik) $n > 0$.
(Wskazówka:) Zajrzyjcie do notatek z wykładu...
(Przykład)

```
- sum 10;
val it = 55 : int
```

1.1.2. *Silnia* `fac`. (2 punkty)
(Typ) `fac: int ->int`
(Opis) `fac` n zwraca $\prod_{i=1}^{n} i$.
(Niezmiennik) $n > 0$.
(Wskazówka) Poprawcie Wasz kod dla `sum`...

1.1.3. *Ciąg Fibonacciego* `fib`. (1 punkt)
(Typ) `fib: int ->int`
(Opis) `fib` n zwaca `fib` $(n-1)$ + `fib` $(n-2)$ jeśli $n \geq 2$ oraz 1 jeśli $n = 0$ lub $n = 1$.
(Niezmiennik) $n \geq 0$.

1.1.4. *Największy wspólny dzielnik* `gcd`. (2 punkty)
(Typ) `gcd: int * int ->int`
(Opis) `gcd` (m, n) zwraca największy wspólny dzielnik liczb m i n obliczony za pomocą algorytmu Euklidesa.
(Niezmiennik) $m \geq 0$, $n \geq 0$, $m + n > 0$.

1.1.5. *Maksimum* `max` *z listy.* (2 punkty)
(Typ) `max: int list ->int`
(Description) `max` l zwraca najwiekszą liczbę z listy l. Jeśli lista jest pusta, zwraca 0.
(Przykład) `max [5,3,6,7,4]` zwraca `7`.

1.2. Funkcje na drzewach binarnych.

1.2.1. *Suma* `sumTree` *liczb naturalnych przechowywanych w drzewie binarnym.* (2 punkty)
(Typ) `sumTree : int tree ->int`
(Opis) `sumTree` t zwraca sumę liczb naturalnych przechowywanych w drzewie t
(Przykład) `sumTree (Node (Node (Leaf 1, 3, Leaf 2), 7, Leaf 4))` zwraca `17`.

1.2.2. *Głębokość* `depth` *zagnieżdżenia drzewa.* (2 punkty)
(Typ) `depth : 'a tree ->int`
(Opis) `depth` t zwraca długość najdłuższej ścieżki od korzenia do liścia.
(Przykład) `depth (Node (Node (Leaf 1, 3, Leaf 2), 7, Leaf 4))` zwraca `2`

1.2.3. *Szukanie* `binSearch` *elementu w drzewie binarnym.* (2 punkty)
(Typ) `binSearch : int tree ->int ->bool`
(Opis) `binSearch` t x zwraca `true` jeżeli element x znajduje się w drzewie t oraz `false` w przeciwnym wypadku.
(Niezmiennik) t jest przeszukiwalnym drzewem binarnym: wszystkie liczby w lewym poddrzewie są mniejsze niż liczba w korzeniu, a wszystkie liczby w prawym poddrzewie są większe niż liczba w korzeniu. Ponadto żadne dwie liczby w drzewie nie są takie same.
(Przykład) `binSearch (Node (Node (Leaf 1, 2, Leaf 3), 4, Leaf 7)) 2` zwraca `true`. `binSearch (Node (Node (Leaf 1, 2, Leaf 3), 4, Leaf 7)) 5` zwraca `false`.

1.2.4. *Przejście* `preorder` *drzewa typu pre-order.* (2 punkty)
(Typ) `preorder: 'a tree ->'a list`
(Opis) `preorder` t zwraca listę elementów wyprodukowaną przez przejście drzewa t typu pre-order.
(Przykład) `preorder (Node (Node (Leaf 1, 3, Leaf 2), 7, Leaf 4))` zwraca `[7, 3, 1, 2, 4]`.
(Uwagi) Przejście drzewa typu pre-order jest jednym z trzech popularnych rekursywnych przejść przez drzewa zdeterminowanych przez ich głębokość. Więcej informacji znaleźć można na przykład tutaj:

`http://webdocs.cs.ualberta.ca/~holte/T26/tree-traversal.html`

1.3. **Funkcje na listach liczb całkowitych.**

1.3.1. *Dodawanie* `listAdd` *każdej pary liczb całkowitych z dwóch drzew.* (2 punkty)
(Typ) `listAdd: int list ->int list ->int list`
(Opis) `listAdd [a,b,c,...] [x,y,z,...]` zwraca `[a+x,b+y,c+z,...]`.
(Przykład) `listAdd [1, 2] [3, 4, 5]` zwraca `[4, 6, 5]`.

1.3.2. *Wkładanie* `insert` *elementu w listę posortowaną.* (2 punkty)
(Typ) `insert: int ->int list ->int list`
(Opis) `insert` m l wkłada m w listę posortowaną l. Lista na wyjściu również musi być posortowana.
(Niezmiennik) Lista l jest posortowana w rosnącym porządku.
(Przykład) `insert 3 [1, 2, 4, 5]` zwraca `[1, 2, 3, 4, 5]`.

1.3.3. *Układanie* `insort` *listy w posortowaną listę.* (2 punkty)
(Typ) `insort: int list ->int list`
(Opis) `insort` l posortowaną listę elementów z listy l.
(Przykład) `insort [3, 7, 5, 1, 2]` zwraca `[1, 2, 3, 5, 7]`.
(Wskazówka) Wykorzystajcie funkcję `insert`

1.4. **Funkcje wyższego rzędu.**

1.4.1. *Składanie* `compose` *dwóch funkcji.* (2 punkty)
(Typ) `compose: ('a ->'b) ->('b ->'c) ->('a ->'c)`
(Opis) `compose` f g zwraca $g \circ f$.
(Uwagi) Wolno Wam użyć tylko dwóch argumentów do zaimplementowania funkcji `compose`. Innymi słowy, coś takiego:

```
fun compose f g = ...
```

jest dopuszczalne, ale już coś takiego:

```
fun compose f g x = ...
```

nie jest.

1.4.2. *"Currowanie"* `curry`. (2 punkty)
(Typ) `curry: ('a * 'b ->'c) ->('a ->'b ->'c)`
(Opis) Zawsze możemy wybrać w jaki sposób chcemy zapisywać funkcję dwóch lub więcej zmiennych. Funkcje są w postaci "scurrowanej" jeśli biorą argumenty po jednym za każdym razem oraz w postaci "niescurrowanej", gdy biorą argumenty jako pary. Funkcja `curry` przerabia funkcję "niescurrowaną" w "scurrowaną".
(Przykład)
`fun multiply x y = x * y (* funkcja scurrowana *)`
`fun multiplyUC (x, y) = x * y (* funkcja niescurrowana *)`
Zastosowanie `curry` do `multiplyUC` daje `multiply`.

1.4.3. *"Odcurrowanie"* `uncurry`. (2 punkty)
(Typ) `uncurry: ('a ->'b ->'c) ->('a * 'b ->'c)`
(Opis) Patrz powyżej.
(Przykład) Zastosowanie `uncurry` do `multiply` daje `multiplyUC`.

1.4.4. `multifun` *wywołujące daną funkcję* n *razy.* (2 punkty)
(Typ) `multifun : ('a ->'a) ->int ->('a ->'a)`
(Opis) `(multifun f n) x` zwraca $\underbrace{\texttt{f}(\texttt{f}(\ldots\texttt{f}(\texttt{x})))}_{n \text{ razy}}$
(Przykład) `(multifun (fn x => x + 1) 3) 1` zwraca `4`.
`(multifun (fn x => x * x) 3) 2` zwraca `256`.
(Niezmiennik) $n \geq 1$.
(Uwagi) Tak jak w przypadku `compose`, wolno Wam użyć tylko dwóch argumentów.

1.5. **Funkcje na liście** `'a list`**.**

1.5.1. *Funkcja* `ltake` *biorąca listę pierwszych* i *elementów listy* l. (2 punkty)
(Typ) `ltake: 'a list ->int ->'a list`
(Opis) `ltake` l n zwraca listę pierwszych n elementów l. Jeśli $n > l$, to zwraca l.
(Przykład) `ltake [3, 7, 5, 1, 2] 3` zwraca `[3,7,5]`.
`ltake [3, 7, 5, 1, 2] 7` zwraca `[3,7,5,1,2]`.
`ltake ["s","t","r","i","k","e","r","z" ] 5` zwraca `["s","t","r","i","k"]`.

1.5.2. *Badanie listy* `lall`. (2 punkty)
(Typ) `lall : ('a ->bool) ->'a list ->bool`
(Opis) `lall` f l zwraca `true` jeśli dla każdego elementu x listy l, fx zwraca `true`; w przeciwnym razie zwraca `false`.
(Przykład) `lall (fn x =>x >0) [1, 2, 3]` zwraca `true`.
`lall (fn x =>x >0) [ 1, 2, 3]` zwraca `false`.

1.5.3. *Funkcja* `lmap` *konwertująca jedną listę w drugą.* (3 punkty)
(Typ) `lmap : ('a ->'b) ->'a list ->'b list`
(Opis) `lmap` f l stosuje f do każdego elementu z l od lewej do prawej, zwracając rezultaty w liście.
(Przykład) `lmap (fn x => x + 1) [1, 2, 3]` zwraca `[2, 3, 4]`.

1.5.4. *Funkcja* `lrev` *odwracająca listę.* (3 punkty)
(Typ) `lrev: 'a list ->'a list`
(Opis) `lrev` l odwraca l.
(Przykład) `lrev [1, 2, 3, 4]` zwraca `[4, 3, 2, 1]`.

1.5.5. *Funkcja* `lzip` *dobierająca w pary odpowiadające sobie elementy dwóch list.* (3 punkty)
(Type) `lzip: ('a list * b' list) ->('a * 'b) list`
(Opis) `lzip` $([x_1, \ldots, x_n], [y_1, \ldots, y_n]) \Rightarrow [(x_1, y_1), \ldots, (x_n, y_n)]$. Jeśli listy różnią się długością, dodatkowe elementy są ignorowane.
(Przykład) `lzip (["Rooney","Park","Scholes","C.Ronaldo"],[8,13,18,7,10,12])` zwraca `[("Rooney"`

1.5.6. *Funkcja* `split` *rozdzielająca listę na dwie.* (3 punkty)
(Typ) `split: 'a list ->'a list * 'a list`
(Opis) `split` l zwraca dwie listy. Pierwsza składa się z elementów, które znajdowały się na nieparzystych miejscach, druga – na parzystych. Dla pustej listy `split` zwraca `([ ], [ ])`. Dla listy jednoelementowej `[x]`, `split` zwraca `([x], [ ])`.
(Przykład) `split [1, 3, 5, 7, 9, 11]` zwraca `([1, 5, 9], [3, 7, 11])`.

1.5.7. *Funkcja* `cartprod` *generująca z dwóch list ich iloczyn kartezjański.* (3 punkty).
(Typ) `cartprod: 'a list ->'b list ->('a * 'b) list`
(Opis) `cartprod` S T zwraca zbiór wszystkich par (x, y) gdzie $x \in S$ oraz $y \in T$. Kolejność wyrazów ma znaczenie.
`cartprod` $[x_1, \ldots, x_n]$ $[y_1, \ldots, y_n] \Rightarrow [(x_1, y_1), \ldots, (x_1, y_n), (x_2, y_1), \ldots, (x_n, y_n)]$.
(Przykład) `cartprod [1, 2] [3, 4, 5]` $\Rightarrow$ `[(1,3), (1,4), (1,5), (2,3), (2,4), (2,5)]`.

2. Część druga

Po tej rozgrzewce możecie przystąpić do rozwiązywania drugiej części zadania. W dalszym ciągu zajmować będziemy się implementacją funkcji, jedyna różnica polega na tym, że teraz będziecie mogli (o ile będziecie chcieli) korzystać z SML/NJ Basic Library.

2.1. **Funkcje rekursywne.**

2.1.1. *Funkcja* `lconcat` *łącząca listę list.* (3 punkty)
(Typ) `lconcat : 'a list list ->'a list`
(Opis) `lconcat` l łączy wszystkie elementy l.
(Przykład) `lconcat [[1, 2, 3], [6, 5, 4], [9]]` zwraca `[1, 2, 3, 6, 5, 4, 9]`.

2.1.2. *Funkcja* `lfoldl` *do zwijania listy od lewej.* (3 punkty)
(Typ) `lfoldl: ('a * 'b ->'b) ->'b ->'a list ->'b`
(Opis) `lfoldl` f e l bierze e oraz pierwszy element l i wykonuje na nich f, a następnie feeduje funkcję otrzymanym rezultatem i drugim argumentem i tak dalej.
`lfoldl` f e $[x_1, x_2, \ldots, x_n]$ zwraca $f(x_n, \ldots, f(x_2, f(x_1, e))\ldots)$ lub e jeśli lista jest pusta.
(Uwagi) Nie wolno Wam używać `List.foldl`

2.2. **Funkcje rekursywne "ogonowo".** Dla każdego z podanych poniżej opisów podajcie implementację rekursywną "ogonowo". We wszystkich przypadkach za wyjątkiem `union` będziecie chcieli wprowadzić rekursywną "ogonowo" funkcję pomocniczą; główna funkcja nie jest rekursywna, ale wywołuje funkcję pomocniczą z odpowiednimi argumentami. Przykładowo implementacja rekursywna "ogonowo" funkcji `sumList` może wyglądać tak:

```
fun sumList inputList =
let
  fun sumList' l accum =
    ...
in
  sumList' inputList 0
end
```

gdzie `sumList'` jest rekursywna "ogonowo".
W przypadku funkcji `union` możecie zechcieć wprowadzić lokalne funkcje pomocnicze. Główna funkcja sama jest rekursywna "ogonowo" i może wykorzystywać owe funkcje pomocnicze.

2.2.1. *Funkcja silnia* `fact`. (1 punkt)
(Typ) `fact: int ->int`
(Opis) `fact` n zwraca $\prod_{i=1}^{n} i$.
(Niezmiennik) $n \geq 0$.

2.2.2. *Funkcja potęgowa* `power`. (1 punkt)
(Typ) `power: int ->int ->int`
(Opis) `power` x n zwraca x^n.
(Niezmiennik) $n \geq 0$.

2.2.3. *Ciąg Fibonacciego* `fib`. (1 punkt)
(Typ) `fib: int ->int`
(Opis) `fib` n zwaca `fib` $(n-1)$ + `fib` $(n-2)$ jeśli $n \geq 2$ oraz 1 jeśli $n = 0$ lub $n = 1$.
(Niezmiennik) $n \geq 0$.
(Wskazówka) Może warto byłoby zastosować pomysł z programowania dynamicznego...?

2.2.4. *Funkcja* `lfilter` *filtrująca listę.* (1 punkt)
(Typ) `lfilter : ('a ->bool) ->'a list ->'a list`
(Opis) `lfilter` p l zwraca wszystkie elementy l które spełniają predykat p.
(Przykład) `lfilter (fn x =>x >2) [0, 1, 2, 3, 4, 5]` zwraca `[3, 4, 5]`.

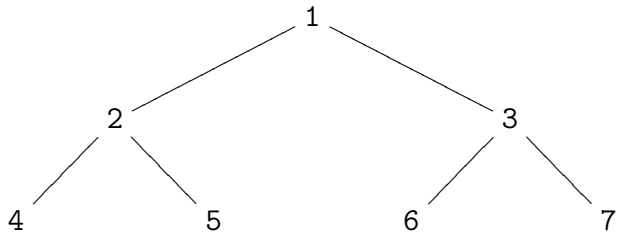
2.2.5. `ltabulate`. (1 punkt)
(Typ) `ltabulate : int ->(int ->'a) ->'a list`
(Opis) `ltabulate` n f stosuje f do każdego z elementów listy `[0, 1, ..., n-1]`.
(Przykład) `ltabulate 4 (fn x =>x * x)` zwraca `[0, 1, 4, 9]`.
(Niezmiennik) $n \geq 0$

2.2.6. *Funkcja* `union` *sumy mnogościowej dwóch zbiorów.* (3 punkty)
(Typ) `union: ''a list ->''a list ->''a list`
(Opis) `union` S T zwraca zbiór, który zawiera wszystkie elementy zbiorów S i T bez powtarzania tych samych elementów. Należy zwrócić uwagę, że wszystkie elementy list mają typ równościowy zgodny z tym, co deklarujemy przez zmienną `''a`. Kolejność elementów listy na wyjściu nie ma znaczenia.
(Niezmiennik) Każdy ze zbiorów zawiera elementy parami różne.
(Przykład) `union [1, 2, 3] [2, 4, 6]` zwraca `[3, 1, 2, 4, 6]`.

2.2.7. *Funkcja* `inorder` *przejścia drzewa binarnego typu inorder.* (4 punkty)
(Typ) `inorder: 'a tree ->'a list`
(Opis) `inorder` t zwraca listę elementów wygenerowaną przez przejście drzewa binarnego t typu inorder.
(Przykład) `inorder (Node (Node (Leaf 1, 3, Leaf 2), 7, Leaf 4))` zwraca `[1, 3, 2, 7, 4]`.
(Przykład) `inorder` może być zaimplementowane następująco:

```
fun inorder t =
  let
    fun inorder' (t' : 'a tree) (post : 'a list) : 'a list = ...
  in
    inorder' t [ ]
  end
```

`post` będzie listą elementów, które trzeba będzie dodać do wyniku przejścia drzewa `'t` typu inorder. Na przykład, gdy `inorder'` odwiedza węzeł oznaczony przez `2` w poniższym drzewie, `post` sprowadzi się do `[1, 6, 3, 7]`.



2.2.8. *Funkcja* `postorder` *przejścia drzewa binarnego typu postorder.* (4 punkty)
(Typ) `postorder: 'a tree ->'a list`
(Opis) `postorder` t zwraca listę elementów wygenerowanych przez przejście drzewa binarnego t typu postorder
(Przykład) `postorder (Node (Node (Leaf 1, 3, Leaf 2), 7, Leaf 4))` zwraca `[1, 2, 3, 4, 7]`.

2.2.9. *Funkcja* `preorder` *przejścia drzewa binarnego typu preorder.* (4 punkty)
(Typ) `preorder: 'a tree ->'a list`
(Opis) `preorder` t zwraca listę elementów wygenerowanych przez przejście drzewa binarnego t typu preorder
(Przykład) `preorder (Node (Node (Leaf 1, 3, Leaf 2), 7, Leaf 4))` zwraca `[7, 3, 1, 2, 4]`.

2.3. **Sortowanie w porządku wstępującym.**

2.3.1. *Funkcja* `quicksort` *do szybkiego sortowania.* (4 punkty)
(Typ) `quicksort: int list ->int list`
(Opis) `quicksort` l implementuje algorytm szybkiego sortowania poprzez wybranie pierwszego elementu listy l jako piwota.
(Przykład) `quicksort [3, 7, 5, 1, 2]` wybiera `3` jako piwota i daje dwie listy `[1, 2]` i `[5, 7]`, które muszą być posortowane niezależnie.

2.3.2. *Funkcja* `mergesort` *łącząca sortowania.* (4 punkty)
(Typ) `mergesort: int list ->int list`
(Opis) `mergesort` l dzieli l na dwie podlisty, sortuje każdą z nich, a następnie łączy dwie posortowane listy. Jeśli l jest o parzystej długości, obydwie podlisty są tej samej długości. W przeciwnym razie jedna podlista jest o jeden element dłuższa od drugiej.

2.4. **Struktury.** W tym fragmencie zadania domowego zapoznamy się z **programowaniem modularnym** w SML-u – ze strukturami i sygnaturami. W pierwszej kolejności zaimplementujemy strukturę dla stosów. Powinniście pamiętać, że ta struktura danych nie jest zwykłą strukturą danych stosu. Najlepiej będzie myśleć o niej jak o mechanizmie dynamicznej alokacji pamięci – patrz wyjaśnienia poniżej.
Uwaga: w sygnaturach `HEAP` oraz `DICT`, `empty` jest typu `unit ->''a heap` oraz `unit ->'a dict`, odpowiednio. Oczywiście byłoby bardziej elegancko użyć typów `'a heap` oraz `''a dict`, ale wtedy trudniej byłoby sprawdzać Wasze zadania...

2.4.1. *Struktura* `Heap` *dla stosów.* (5 punktów)
Struktura `Heap` odpowiada sygnaturze `HEAP`. Stos jest mechanizmem dynamicznej alokacji pamięci.

```
signature HEAP =
sig     exception InvalidLocation
  type loc
  type 'a heap
  val empty : unit ->'a heap
  val allocate : 'a heap ->'a ->'a heap * loc
  val dereference : 'a heap ->loc ->'a
  val update : 'a heap ->loc ->'a ->'a heap
end
```

- `loc` jest wewnętrzną reprezentacją lokacji, która jest trochę podobna do *pointer* w C. `type loc` nie jest widoczne poza strukturą;
- `'a heap` jest stosem dla typu `'a`;
- `empty ()` zwraca pusty stos;
- `allocate` h v alokuje daną wartość v do nowej komórki stosu i zwraca parę (h', l) zmodyfikowanego stosu h' i lokacji l tej komórki;

- `dereference` $h\ l$ bierze wartość v przechowywaną w komórce stosu o lokalizacji l; `InvalidLocation` jest zgłaszane, gdy l ma nieprawidłową `loc`;
- `update` $h\ l\ v$ zmienia komórkę stosu o lokalizacji l z daną wartością v i zwraca zmieniony stos h'; `InvalidLocation` jest zgłoszone gdy l ma nieprawidłową `loc`.

2.4.2. *Sygnatura* `DICT`. `DICT` jest sygnaturą dla słowników.

```
signature DICT =
sig
  type key
  type 'a dict
  val empty : unit ->'a dict
  val lookup : 'a dict ->key ->'a option
  val delete : 'a dict ->key ->'a dict
  val insert : 'a dict ->key * 'a ->'a dict
end
```

- `empty ()` zwraca pusty słownik;
- `lookup` $d\ k$ szuka klucza k w słowniku d; jeśli klucz zostanie znaleziony, zwraca dołączoną do niego zawartość, w przeciwnym wypadku zwraca `NONE`.
- `delete` $d\ k$ kasuje klucz k i dołączoną do niego zawartość w słowniku d i zwraca tak otrzymany słownik d'; jeśli dany klucz nie znajduje się w słowniku d, zwraca dany słownik bez zmian;
- `insert` $d\ (k, v)$ wkłada nowy klucz k i dołączoną do niego zawartość v do słownika d; jeśli klucz k występuje już w słowniku d, po prostu nadpisuje zawartość zgodnie z nową wartością v.

2.4.3. *Struktura* `DictList`. (5 punktów)

Zaimplementujcie strukturę `DictList` o sygnaturze `DICT` z definicją `'a dict = (key * 'a) list`.

Struktura `DictList` używa listy par jako reprezentacji słownika. Implementacja powinna być bezpośrednia, jako że o liście par możemy myśleć jak o słowniku.

2.4.4. *Struktura* `DictFun`. (5 punktów)

Zaimplementujcie strukturę `DictFun` o sygnaturze `DICT` z definicją `'a dict = key ->'a option`.

Struktura `DictFun` używa "reprezentacji funkcyjnej" słowników. Pomysł polega na tym, że reprezentujemy dany słownik jako funkcję, która, przy danym kluczu, zwraca dołączoną do niego zawartość. Zaimplementowanie struktury `DictFun` może być albo bardzo trudne, albo zupełnie proste, w zależności od tego, czy podejdziecie do problemu myśląc "imperatywnie", czy "funkcyjnie". Dobra rady Wujka Dobra Rada jest następująca: zapomnijcie o wszystkim, czego nauczyliście się o programowaniu imperatywnym i po prostu myślcie funkcjonalnie – sami zobaczycie, jak bardzo będziecie zaskoczeni zwięzłością i elegancją napisanego przez Was kodu.