Zadanie I: SML - Type Checker

Rozważamy następujący typ dla reprezentacji lambda termów:

```
type label = string
datatype lterm = Number of (IntInf.int)
                        |Label of label
                        |App of lterm*lterm
                        |Abs of label*lterm
                        |Let of ((label*lterm) list) * lterm
```

oraz następujący do reprezentacji typów:

```
type label = string
datatype TTerm = VAR of label
               | ARR of TTerm * TTerm
               | LIST of TTerm
               | INT
```

(`ARR (t1,t2)` reprezentuje typ funkcyjny `t1->t2`). Przy czym zakładamy dodatkowo że `label` może się składać wyłącznie ze znaków spełniających predykat `Char.isAlphaNum`.

Zadanie polega na implementacji type checkera, który znajdzie dla podanego termu najbardziej ogólny typ w dostarczonym kontekście. Type checker powinien być funktorem parametryzowanym strukturą implementującą unifikator termów pierwszego rzędu. Termy pierwszego rzędu reprezentujemy następująco:

```
type name = char list
datatype term = Fun of name * term list | Var of name
type substitution = (name * term ) list (* triangular form !!! *)
```

Typ `substitution` reprezentuje podstawienie w tzw. formie trójkątnej. Tzn. aby wykonać podstawienie [(v1,t1), (v2,t2),..., (vk,tk)] na termie T należy najpierw wszystkie wystąpienia v1 w T zastąpić termem t1, potem dla tak skonstruowanego termu należy wszystkie wystąpienia v2 zastąpić termem t2 itd. Sygnatura dla unifikatora wygląda następująco:

```
signature UNIFIER = sig
  val unify: term -> term -> substitution option
  val lunify : (term list * term list) option -> substitution option
end
```

Funkcja `unify` zwraca najbardziej ogólne podstawienie unifikującer dla zadanych termów lub `NONE` jeśli taki nie istnieje. Funkcja `lunify` działa analogicznie na listach par termów (tzn. znajduje najbardziej ogólne postawienie unifikujące wpólne dla wszystkich par).

Szablon funktora:

```
functor TYPECHECK(U:UNIFIER) = struct
        fun typecheck lmterm context=  ...
end
```

Pierwszy argument `lmterm` jest typu `lterm` drugi jest kontekstem opisanym w następnym paragrafie. Funkcja powinna zwrócić wartość typu `TTerm option` reprezentującą najbardziej ogólny typ dla wejściowego termu w zadanym kontekście lub `NONE` jeśli termowi nie da się przypisać typu.

Kontekst:

Kontekst dla type checkera, dostarczony pod nazwą `context`, jest obiektem typu `(string* TTerm) list`. Można przyjąć że wszystkie wolne zmienne termu wejściowego będą miały przypisane typy w kontekście. Typy w kontekście należy traktować jak schematy typów (typy polimorficzne), tzn. należy je zgeneralizować po wszystkich zmiennych. Przykładowy kontekst:

```
local
                infixr 6 -->
                fun a --> b= ARR (a,b)
in
                val context= [
                        ("PLUS",INT --> INT --> INT),
                        ("CONS", VAR "a" --> (LIST (VAR "a")) --> (LIST (VAR "a"))),
                        ("NIL", LIST (VAR "a")),
                        ("hd",(LIST (VAR "a")) --> VAR "a")
                        ]
end
```

System typów

W poniższej specyfikacji wyrażenia dla typów monomorficznych oznaczane są przez τ, τ', τ''. Wyrażenia są budowane zgodnie z poniższa gramatyką (α oznacza zmienną typową):

$$\tau = \alpha \mid \tau' \to \tau'' \mid \text{LIST } \tau' \mid \text{int}$$

Wyrażenia te odpowiadają obiektom typu `TTerm`.

Schematy typów oznaczamy przez $\sigma, \sigma_1, \sigma_2, \ldots$. Schematy są definiowane następująco:

$$\sigma = \tau \mid \forall\ \alpha.\ \sigma$$

Następujące reguły specyfikują bazowy bazowy system typów:

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad [\text{Var}]$$

$$\frac{\Gamma \vdash e_0 : \tau \to \tau' \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0\ e_1 : \tau'} \quad [\text{App}]$$

$$\frac{\Gamma,\ x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda\ x\ .\ e : \tau \to \tau'} \quad [\text{Abs}]$$

$$\frac{\Gamma \vdash e_0 : \sigma \qquad \Gamma,\ x : \sigma \vdash e_1 : \tau}{\Gamma \vdash \text{let } x = e_0 \text{ in } e_1 : \tau} \quad [\text{Let}]$$

$$\frac{\Gamma \vdash e : \sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash e : \sigma} \quad [\text{Inst}]$$

$$\frac{\Gamma \vdash e : \sigma \quad \alpha \notin \text{free}(\Gamma)}{\Gamma \vdash e : \forall\ \alpha\ .\ \sigma} \quad [\text{Gen}]$$

(podstawowa różnica pomiędzy `let` i abstrakcją jest taka, że `let` wprowadza do kontekstu schemat typu, a abstrakcja po prostu typ)

Bazowy system typów rozszerzamy o:

- Możliwość wielu definicji w jednej klauzuli `let`.
- Możliwość rekursywnych definicji w `let` (także wzajemnie rekursywnych dla etykiet definiowanych w obrębie jednej klauzuli). Wystąpienia etykiet definiowanych w klauzuli `let` są monomorficzne w samej definicji i w innych definicjach w obrębie tej samej klauzuli.

Przystępne opracowanie można znaleźć w Simon Peyton Jones, The Implementation of Functional Programming Languages (rozdziały 8 i 9).

Uwagi:

Nazwy zmiennych w zwracanym typie nie mają znaczenia (poza tym że nie powinnyć być bez potrzeby długie).

Przykłady:

Dla danych wejściowych:	Poprawna odpowiedzią jest:
`(fn a => fn b => fn c => (a c) (b c))`	`(v0->v1->v2)->(v0->v1)->v0->v2`

`(fn S => fn K1 => fn K2 => S K1 K2)`
`(fn a => fn b => fn c => (a c) (b c))`
`(fn x=> fn y => x)`
`(fn x=> fn y => x)`
	`v0->v0`

`(fn S => fn K => S K K)`
`(fn a => fn b => fn c => (a c) (b c))`
`(fn x=> fn y => x)`
	`NO TYPE`

`let`
`  fun S a b c = a c (b c)`
`  fun K a b = a`
`in S K K end`
	`v0->v0`

`(fn x=> fn y => x::y)`	`v0->[v0]->[v0]`
`fn y => y + 1`	`int->int`

`(fn I => (let`
`  val x = let val y = I 7`
`  in x end)`
`)(fn x => x)`
	`int`
	`in I x end`

`let`
`  fun foldl f z l1 =IF 0 z (foldl f (f z (hd l1)) (tl l1))`
`  val ones = 1::ones`
`  fun add x y = x+y`
`  fun f1 i = 1+ (f2 i)`
`  fun f2 i = 1+ (f1 i)`
`in  foldl add  end`
	`int->[int]->int`
	`(po dodaniu do kontekstu:`
	`IF:int->a->a->a`
	`tl:[a]->[a])`