Egzamin - Metody Programowania

Suma punktów: 100

Zadanie 1. (6 pkt - 2 pkt za podpunkt)

W poniższych wyrażeniach podkreśl wolne wystąpienia zmiennych. Dla każdego związanego wystąpienia zmiennej, narysuj strzałkę od tego wystąpienia do wystąpienia wiążącego je.

a)

```ocaml
let f x y =
  let g z = x + z in
  g (y + w)
```

b)

```ocaml
fun x ->
  let rec f y =
    if y > 0 then x + f (y - 1)
    else g x
  in f x
```

c)

```ocaml
let x = y + 1 in
let y = x + 2 in
fun z -> x + y + z
```

Zadanie 2. (6 pkt - 2 pkt za podpunkt)

Przeprowadź alpha-konwersję poniższych wyrażeń, zmieniając nazwy wszystkich związanych zmiennych według podanego schematu:

a) Dodaj suffix "_1" do wszystkich związanych zmiennych:

```ocaml
fun x -> let y = x + 1 in fun x -> x + y
```

b) Zmień nazwy używając prefiksu "new_":

```ocaml
let rec f x =
  let g y = f (x + y) in
  g x
```

c) Przemianuj zmienne konfliktujące przy podstawieniu `[z/y]` w:

```ocaml
fun x -> let y = x + z in fun z -> y + z
```

Zadanie 3. (8 pkt - 2 pkt za podpunkt)

Podaj najogólniejszy typ poniższych wyrażeń lub napisz "BRAK TYPU":

a) `fun f g x -> g (f x x)`

b) `fun x -> (x 1, x true, x [])`

c) `let rec f x = f (f x) in f`

d) `fun f -> let g x = f (x, x) in g`

Zadanie 4. (6 pkt - 2 pkt za podpunkt)

Napisz DOWOLNĄ funkcję realizującą podany typ:

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

b) `('a -> 'a -> bool) -> 'a list -> 'a list list`

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

Zadanie 5. (8 pkt - 1 pkt za podpunkt)

Zaimplementuj poniższe funkcje używając `fold`:

a) `sum_of_squares : int list -> int` - suma kwadratów elementów

b) `filter : ('a -> bool) -> 'a list -> 'a list` - filtrowanie elementów

c) `partition : ('a -> bool) -> 'a list -> ('a list * 'a list)` - podział na spełniające i niespełniające

d) `map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list` - mapowanie po dwóch listach

e) `take_while : ('a -> bool) -> 'a list -> 'a list` - bierze elementy dopóki spełniają predykat

f) `index_of : 'a -> 'a list -> int option` - indeks pierwszego wystąpienia

g) `unique : 'a list -> 'a list` - usuwa duplikaty (zachowuje pierwsze wystąpienie)

h) `group_by : ('a -> 'b) -> 'a list -> ('b * 'a list) list` - grupuje elementy według klucza

Zadanie 6. (4 pkt - 1 pkt za podpunkt)

Dla typu `type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree`, zaimplementuj używając `fold_tree`:

```ocaml
let rec fold_tree f acc = function
  | Leaf -> acc
  | Node (l, v, r) -> f (fold_tree f acc l) v (fold_tree f acc r)
```

a) `tree_sum : int tree -> int` - suma elementów

b) `tree_height : 'a tree -> int` - wysokość drzewa

c) `tree_map : ('a -> 'b) -> 'a tree -> 'b tree` - mapowanie drzewa

d) `tree_filter : ('a -> bool) -> 'a tree -> 'a list` - lista elementów spełniających predykat

Zadanie 7. (4 pkt - 2 pkt za podpunkt)

Dla typu `type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree`, zaimplementuj:

a) `mirror : 'a tree -> 'a tree` - lustrzane odbicie drzewa

b) `paths_to_leaves : 'a tree -> 'a list list` - wszystkie ścieżki od korzenia do liści

Zadanie 8. (8 pkt - 4 pkt za podpunkt)

Rozważ funkcje:

```ocaml
let rec append xs ys = match xs with
  | [] -> ys
  | h::t -> h :: append t ys

let rec rev_append xs ys = match xs with
  | [] -> ys
  | h::t -> rev_append t (h::ys)
```

a) Udowodnij indukcyjnie: `rev_append xs ys = (rev xs) @ ys`

b) Udowodnij indukcyjnie: `length (append xs ys) = length xs + length ys`

Zadanie 9. (4 pkt - 2 pkt za podpunkt)

a) Napisz gramatykę bezkontekstową dla języka wyrażeń arytmetycznych z priorytetami:

- operatory: `+`, `*`, `-` (unarny)
- nawiasy: `(`, `)`
- liczby: sekwencje cyfr
- `*` ma wyższy priorytet niż `+`
- `-` unarny ma najwyższy priorytet

b) Czy Twoja gramatyka jest jednoznaczna? Jeśli nie, podaj przykład niejednoznaczności.

Zadanie 10. (6 pkt - 2 pkt za podpunkt)

Chcemy zaimplementować interpreter języka z:

- funkcjami pierwszej klasy
- leniwą ewaluacją (call-by-need)
- nieskończonymi strukturami danych

a) Zaproponuj typ `value` dla tego języka

b) Jak reprezentować thunki (opóźnione obliczenia)?

c) Jak zapewnić, że raz obliczona wartość nie będzie obliczana ponownie?

Zadanie 11. (8 pkt)

Zaimplementuj fragment interpretera używając monady State do śledzenia liczby wykonanych redukcji:

ocaml

```ocaml
type 'a state = int -> ('a * int)

let return x = fun s -> (x, s)
let bind m f = fun s -> let (x, s') = m s in f x s'
let get = fun s -> (s, s)
let put s' = fun _ -> ((), s')
let modify f = fun s -> ((), f s)

(* Zaimplementuj: *)
let rec eval_with_count env = function
  | Var x -> ...
  | App (e1, e2) -> ...
  | Fun (x, body) -> ...
  (* Każda redukcja (beta-redukcja) powinna zwiększać licznik *)
```

Zadanie 12. (12 pkt)

Zaimplementuj kompilator wyrażeń do notacji RPN (Reverse Polish Notation):

ocaml

```ocaml
type expr =
  | Num of int
  | Var of string
  | Add of expr * expr
  | Sub of expr * expr
  | Mul of expr * expr
  | Div of expr * expr
  | Let of string * expr * expr
  | If of expr * expr * expr  (* if e1 > 0 then e2 else e3 *)

type rpn_instr =
  | Push of int
  | Load of string
  | Store of string
  | Add | Sub | Mul | Div
  | Dup           (* duplikuj szczyt stosu *)
  | Swap          (* zamień 2 elementy na szczycie *)
  | Drop          (* usuń szczyt *)
  | JumpIfNotPositive of int  (* skok jeśli <= 0 *)
  | Jump of int

let rec compile (env : string list) (e : expr) : rpn_instr list = ...
```

Wskazówka: `env` przechowuje nazwy zmiennych w kolejności na stosie.

Zadanie 13. (10 pkt)

Rozważmy język z mutowalnymi referencjami i funkcjami wyższego rzędu:

```ocaml
type expr =
  | Int of int
  | Bool of bool
  | Var of string
  | Fun of string * typ option * expr  (* opcjonalna anotacja typu *)
  | App of expr * expr
  | Let of string * expr * expr
  | Ref of expr                (* utworzenie referencji *)
  | Deref of expr              (* odczytanie referencji *)
  | Assign of expr * expr      (* przypisanie do referencji *)
  | Seq of expr * expr         (* sekwencja e1; e2 *)
  | If of expr * expr * expr
  | Equal of expr * expr       (* porównanie strukturalne *)
```

a) (3 pkt) Uzupełnij definicję typów:

```ocaml
type typ =
  | TInt
  | TBool
  | TFun of typ * typ
  | TRef of ___
```

b) (7 pkt) Napisz prosty type checker (bez inferencji - Fun musi mieć anotację):

```ocaml
type tenv = (string * typ) list

let rec type_check (env : tenv) (e : expr) : typ = ...
```