JPPML

Jonasz Aleszkiewicz

1. WSTĘP

JPPML to język funkcyjny, inspirowany przede wszystkim SML-em i OCamlem, ale też trochę Haskellem.

Ze względów estetycznych, czcionka w tym dokumencie wyświetla == jako ═, != jako ≠, -> jako →.

2. OPIS JĘZYKA

2.1. TYPY

```
let u (* unit *)              = ()
let x (* int *)               = 1
let b (* bool *)              = True
let s (* string *)            = "Hello!"
let l (* int list *)          = [1, 2, 3]
let t (* int * string * int *) = (1, "a", 3)
let f (* 'a → int *)          = fn x → 1
```

Nie ma typów `float` ani `char`. Typy `bool` i `list` nie są wbudowane tylko zdefiniowane w bibliotece standardowej, ale istnieje wbudowany cukier syntaktyczny dla list. Int jest liczbą całkowitą ze znakiem. Bool przyjmuje wartości `True` i `False` (wielką literą). Nie ma rekordów.

Są typy polimorficzne w stylu ML, uzgadniane podczas kompilacji.

2.2. OPERATORY

Następujący podzbiór operatorów z OCamla [1] (kolejność według priorytetu, po prawej odpowiadające wbudowane funkcje i typy funkcji odpowiadającej operatorom):

```
-x          __neg                          int → int
f x         (wywołanie funkcji)
```

```
* /          __mul __div                      int → int → int
+ -          __add __sub                      int → int → int
::           __cons                           'a → 'a list → 'a list
@ ^          __append __cat                   'a list → 'a list → 'a list
                                                  string → string → string
== !=        __eq __neq                       'a → 'a → bool
< <= > >=    __lt __gt __le __ge              int → int → bool
&&           (koniunkcja, leniwa) __and       bool → (unit → bool) → bool
||           (alternatywa, leniwa) __or       bool → (unit → bool) → bool
if           __if                       bool → (unit → 'a) → (unit → 'a) → 'a
let case fn
```

Łączność jak w OCamlu. Nie ma definiowania własnych operatorów. Nie ma składni typu `(+)` żeby dostać funkcje odpowiadającą operatorowi. Operatory == i != (== i !=) jak w OCamlu są specjalne i działają na każdych typach, nie dałoby się ich zdefiniować w kodzie użytkownika. Nie ma operatora `;`, do sekwencjonowania poleceń można użyć `ignore [a, b]` lub `let _ = a in b`.

2.3. KOMENTARZE

```
-- Jednoliniowy
(* Wielo-
   liniowy *)
```

Komentarze jednoliniowe jak w Haskellu; wieloliniowe jak w OCamlu. Nie mogą być zagnieżdżone.

2.4. WYRAŻENIE WARUNKOWE

```
if a then b else c
```

2.5. PRZYPISANIA

```
let a = 24                    -- jako deklaracja
let _ = 7                     -- ignorowanie wartości
let _ =
  let a = 41 in a + 1         -- jako wyrażenie
let b = a + 1 and a = 41      -- kilka przypisań
let ones = 1 :: ones          -- rekurencyjne przypisanie

let a = [] in (1 :: a, "s" :: a)  -- uzgodnienie typów po przypisaniu

let x = 2
let x = 3       -- okej, przysłanianie
let x = x + 1   -- nie okej
```

Nie można użyć patternów w `let` oprócz _ oznaczającego zignorowanie wartości. Nie ma składni do eksplicytnego ustalania typów (`let x : int = 0`). Wszystkie przypisania są rekurencyjne (działają jak `let rec` w OCamlu). Można zasłaniać inne nazwy.

2.6. FUNKCJE

```
let inc = fn x → x + 1          -- funkcja
let f = fn x → f (x - 1)        -- funkcja rekurencyjna
let plus = fn a b → a + b       -- funkcja wieloargumentowa
let plus = fn a → fn b → a + b
```

```
let plus1 = plus 1                -- częściowa aplikacja
let _ = 0                         -- odrzucenie wyniku

let _ = map (fn x → x + 1) l      -- funkcja anonimowa
```

Nie ma nazwanych ani opcjonalnych argumentów. Nie ma składni `let f x = ...`, jest tylko `fn`. Nie można użyć patternów w argumentach funkcji.

2.7. TYPY ALGEBRAICZNE

```
type tree =
  | Oak
  | Ash
  | Forest of tree * tree
let a = Oak
let f = Forest (a, Ash)

-- Typ parametryzowany
type 'a list =
  | Nil
  | Cons of 'a * 'a list
let l = Cons (1, Cons (3, Nil))

-- Wzajemnie rekurencyjne typy
type red_node =
  | Red of blue_node
and blue_node =
  | Blue of red_node
let a = Red b and b = Blue c and c = Red d and d = Blue a

-- Użycie konstruktora jako funkcji
let x = (map Some [1, 2, 3]) @ [None]
```

Nie ma aliasów typów ani rekordów. Typy są rekurencyjne. Każdemu konstruktorowi (np. `Some`, `None`) odpowiada funkcja lub wartość. Typów ani konstruktorów nie można przykrywać.

2.8. WZORCE

```
case x of
  | 2 → ...
  | x → ...
  | _ → ...
case p of
  | Pair (x, y) → ...
case l of
  | head :: tail → ...
  | [1, 2, x] → ...

case x of
  | Some (a, b) → case a of
    | 1 → ...
    | 2 → ...
```

Wzorce występują wyłącznie w konstrukcji case-of. Mogą być dowolnie zagnieżdżone. Nie ma `as`, `when`, `exception`, wielu wzorców dla jednego przypadku (`case x of a | b → ...`). Nie ma

sprawdzania kompletności pattern matchingu podczas kompilacji. Kiedy wartość nie odpowiada żadnemu przypadkowi jest to błąd wykonania.

2.9. WYJĄTKI

```
exception ParseExn
exception ParseExn of int * int
exception A and B
let _ = raise (ParseExn (row, col))
let _ = failwith "oops :c"
```

Nie ma łapania wyjątków, zawsze zatrzymują działanie programu. Parametry wyjątków są używane tylko do wypisania błędu.

2.10. MODUŁY

```
open List
```

Żeby użyć funkcji z niektórych modułów, trzeba użyć `open`. Nie ma kwalifikowanego importowania. Można importować tylko wbudowane moduły (`List` oraz `Option`).

2.11. WBUDOWANE FUNKCJE

Biblioteka standardowa jest oparta na bibliotece standardowej OCamla.

Ponieważ nie ma kwalifikowanego importowania, niektóre funkcje nazywają się inaczej niż w OCamlu aby uniknąć konfliktów.

Oto funkcje zawarte w bibliotece standardowej:

```
Assert_failure : __exn = Assert_failure
Division_by_zero : __exn = Division_by_zero
Err : 'a → ('b, 'a) result = <fn>
Exit : __exn = Exit
Failure : string → __exn = <fn>
False : bool = False
Invalid_argument : string → __exn = <fn>
Match_failure : __exn = Match_failure
None : 'a option = None
Not_found : __exn = Not_found
Ok : 'a → ('a, 'b) result = <fn>
Some : 'a → 'a option = <fn>
True : bool = True
__Cons : 'a * 'a list → 'a list = <fn>
__Nil : 'a list = []
__add : int → int → int = <fn>
__and : bool → (unit → bool) → bool = <fn>
__append : 'a list → 'a list → 'a list = <fn>
__cat : string → string → string = <fn>
__cons : 'a → 'a list → 'a list = <fn>
__div : int → int → int = <fn>
__eq : 'a → 'a → bool = <fn>
__ge : int → int → bool = <fn>
__gt : int → int → bool = <fn>
__if : bool → (unit → 'a) → (unit → 'a) → 'a = <fn>
```

```
__le : int → int → bool = <fn>
__lt : int → int → bool = <fn>
__mul : int → int → int = <fn>
__ne : 'a → 'a → bool = <fn>
__neg : int → int = <fn>
__or : bool → (unit → bool) → bool = <fn>
__sub : int → int → int = <fn>
__try : (unit → unit) → string option = <fn>
abs : int → int = <fn>
assert : bool → unit = <fn>
bool_of_string : string → bool option = <fn>
cat : string → string → string = <fn>
combine : ('a → 'b) → ('c → 'a) → 'c → 'b = <fn>
const : 'a → 'b → 'a = <fn>
curry : ('a → 'b → 'c) → 'a * 'b → 'c = <fn>
failwith : string → 'a = <fn>
flip : ('a → 'b → 'c) → 'b → 'a → 'c = <fn>
fst : 'a * 'b → 'a = <fn>
id : 'a → 'a = <fn>
ignore : 'a → unit = <fn>
invalid_arg : string → 'a = <fn>
map : ('a → 'b) → 'a list → 'b list = <fn>
max : int → int → int = <fn>
min : int → int → int = <fn>
mod : int → int → int = <fn>
negate : ('a → bool) → 'a → bool = <fn>
not : bool → bool = <fn>
pred : int → int = <fn>
print : 'a → unit = <fn>
print_endline : string → unit = <fn>
print_int : int → unit = <fn>
print_string : string → unit = <fn>
raise : 'a → 'b = <fn>
snd : 'a * 'b → 'b = <fn>
succ : int → int = <fn>
to_string : 'a → string = <fn>
uncurry : ('a * 'b → 'c) → 'a → 'b → 'c = <fn>
```

W module `List`:

```
append : 'a list → 'a list → 'a list = <fn>
chop : int → 'a list → 'a list = <fn>
combine : 'a list → 'b list → ('a * 'b) list = <fn>
concat : 'a list list → 'a list = <fn>
cons : 'a → 'a list → 'a list = <fn>
exists : ('a → bool) → 'a list → bool = <fn>
find : ('a → bool) → 'a list → 'a = <fn>
flatten : 'a list list → 'a list = <fn>
for_all : ('a → bool) → 'a list → bool = <fn>
hd : 'a list → 'a = <fn>
init : int → (int → 'a) → 'a list = <fn>
is_empty : 'a list → bool = <fn>
iter : 'a → 'b list → unit = <fn>
length : 'a list → int = <fn>
lists_equal : ('a → 'b → bool) → 'a list → 'b list → bool = <fn>
mapi : (int → 'a → 'b) → 'a list → 'b list = <fn>
mem : 'a → 'a list → bool = <fn>
merge : int list → int list → int list = <fn>
```

```
merge_sort : int list → int list = <fn>
nth : 'a list → int → 'a = <fn>
nth_opt : 'a list → int → 'a option = <fn>
partition : 'a → 'b list → 'b list * 'b list = <fn>
rev : 'a list → 'a list = <fn>
rev_append : 'a list → 'a list → 'a list = <fn>
sort : int list → int list = <fn>
take : int → 'a list → 'a list = <fn>
tl : 'a list → 'a list = <fn>
zip : 'a list → 'b list → ('a * 'b) list = <fn>
```

Moduł Option:

```
get : 'a option → 'a = <fn>
value : 'a option → 'a → 'a = <fn>
```

2.12. SKŁADNIA FORMALNA

W formacie LBNF [2] przyjmowanym przez BNFC [3]. Oparta na gramatyce SML [4], i gramatyce Latte [5].

Gramatyka jest niejednoznaczna przy zagnieżdżonych konstrukcjach case-of, zawsze w przypadku początku nowego przypadku rozszerzany jest nagłębszy case-of, czyli np. `case a of 1 → case b of 2 → 3 | 3 → 4` jest równoważne z `case a of 1 → (case b of 2 → 3 | 3 → 4)`.

Gramatyka przyjmuje konstrukcje jak `type int list = A`, które są niedozwolone w gramatyce OCamla. W JPPML są odrzucane na poziomie kontroli typów.

```
entrypoints [Dec]                                   ;

-- Identifiers

token Id (lower (letter | digit | '_')*)            ;
token IdCap (upper (letter | digit | '_')*)         ;
token IdVar ('\'' (letter | digit | '_')*)          ;

-- Constants

CInt.     Con       ::= Integer                     ; -- 42
CString.  Con       ::= String                      ; -- "abc"
CUnit.    Con       ::= "(" ")"                     ; -- ()

-- Expressions

ECon.     Exp11     ::= Con                         ; -- 42
EObjCon.  Exp11     ::= IdCap                       ; -- Empty
EId.      Exp11     ::= Id                          ; -- x

ETup.     Exp11     ::= "(" Exp "," [Exp] ")"       ; -- (a, b)
ELst.     Exp11     ::= "[" [Exp] "]"               ; -- [a, b]
separator Exp ","                                   ;

-- Odpowiada też za konstruowanie obiektów, np. "Some x" będzie zinterpretowane
-- jako EApp (EObjCon "Some") (EId "x").
EApp.     Exp10     ::= Exp10 Exp11                 ; -- f x
```

```
ENeg.     Exp9      ::= "-" Exp10                    ; -- -10

EMul.     Exp8      ::= Exp8 "*" Exp9                ; -- 2 * 2
EDiv.     Exp8      ::= Exp8 "/" Exp9                ; -- 8 / 4

EAdd.     Exp7      ::= Exp7 "+" Exp8                ; -- 2 + 2
ESub.     Exp7      ::= Exp7 "-" Exp8                ; -- 3 - 2
ECons.    Exp6      ::= Exp7 "::" Exp6               ; -- 1 :: [2, 3]
EAppend.  Exp5      ::= Exp6 "@" Exp5                ; -- [1, 2] @ [2, 3]
ECat.     Exp5      ::= Exp6 "^" Exp5                ; -- "ab" ^ "c"

-- Niedozwolone np. a < b < c
ERel.     Exp4      ::= Exp5 ERelOp Exp5             ;
EREq.     ERelOp    ::= "=="                         ;
ERNe.     ERelOp    ::= "/="                         ;
ERLt.     ERelOp    ::= "<"                          ;
ERLe.     ERelOp    ::= "<="                         ;
ERGt.     ERelOp    ::= ">"                          ;
ERGe.     ERelOp    ::= ">="                         ;

EAnd.     Exp3      ::= Exp3 "&&" Exp4               ; -- a && b

EOr.      Exp2      ::= Exp2 "||" Exp3               ; -- a || b

EIf.      Exp1      ::= "if" Exp "then" Exp "else" Exp1 ; -- if a then b else c

ELet.     Exp       ::= "let" [LetBind] "in" Exp     ; -- let ... in ...

ECase.    Exp       ::= "case" Exp "of" [ECaseBind]  ; -- case x of ...
eCaseAlt. Exp       ::= "case" Exp "of" "|" [ECaseBind] ; -- case x of ...
define eCaseAlt a b = ECase a b                      ;
ECBJust.  ECaseBind ::= Pat "->" Exp                 ;
separator nonempty ECaseBind "|"                     ;

EFn.      Exp       ::= "fn" [Id] "->" Exp           ;
separator nonempty Id ""                             ;

coercions  Exp 11                                    ;

-- Patterns

PCon.     Pat3      ::= Con                          ; -- 0
PId.      Pat3      ::= Id                           ; -- x
PWild.    Pat3      ::= "_"                          ; -- _

PTup.     Pat3      ::= "(" Pat "," [Pat] ")"        ; -- (a, b)
PLst.     Pat3      ::= "[" [Pat] "]"                ; -- [a]
separator Pat ","                                    ;

PObjCon.  Pat3      ::= IdCap                        ; -- Empty

PObj.     Pat1      ::= IdCap Pat2                   ; -- Some x

PCons.    Pat       ::= Pat1 "::" Pat                ; -- 1 :: [2, 3]

coercions  Pat 3                                     ;

-- Types
```

```
TIdVar.   Typ3      ::= IdVar                       ; -- 'a
TId.      Typ3      ::= TypLst Id                   ; -- int list

TLEmpty.  TypLst    ::=                             ;
TLOne.    TypLst    ::= Typ3                        ; -- int
TLMany.   TypLst    ::= "(" Typ "," [Typ] ")"       ; -- (int, string)
separator nonempty Typ ","                          ;

TTup.     Typ1      ::= Typ2 "*" [TTupElem]         ; -- int * string
TTupJust. TTupElem  ::= Typ2                        ;
separator nonempty TTupElem "*"                     ;

TFn.      Typ       ::= Typ1 "→" Typ                ; -- int → string

coercions Typ 3                                     ;

-- Declarations

DLet.     Dec       ::= "let" [LetBind]             ; -- let a = 0 and f x = a
DType.    Dec       ::= "type" [TypBind]            ; -- type x = A | B of c
DExn.     Dec       ::= "exception" [ExnBind]       ; -- exception A of string
DOpen.    Dec       ::= "open" [IdCap]              ; -- open List
separator nonempty IdCap "and"                      ;
separator Dec ""                                    ;

LBJust.   LetBind   ::= Id "=" Exp                  ; -- let x = 4
LBAnon.   LetBind   ::= "_" "=" Exp                 ; -- let _ = x
separator nonempty LetBind "and"                    ;

TBJust.   TypBind   ::= TypLst Id "=" [DTag]        ; -- 'a t = A | B
tBJust.   TypBind   ::= TypLst Id "=" "|" [DTag]    ; -- 'a t = | A | B
define tBJust a b c = TBJust a b c                  ;
separator nonempty TypBind "and"                    ;

DTCon.    DTag      ::= IdCap                       ; -- A
DTArg.    DTag      ::= IdCap "of" Typ              ; -- A of int
separator nonempty DTag "|"                         ;

EBCon.    ExnBind   ::= IdCap                       ; -- A
EBArg.    ExnBind   ::= IdCap "of" Typ              ; -- A of int
separator nonempty ExnBind "and"                    ;
```

3. PRZYKŁADOWY KOD

```
open List

let fizzbuzz = fn n →
  let aux = fn x →
    if mod x 15 == 0 then print_endline "fizzbuzz"
    else if mod x 3 == 0 then print_endline "fizz"
    else if mod x 5 == 0 then print_endline "buzz"
    else ()
  and range = fn a b →
    init (b - a) (fn x → a + x)
  in
    iter aux (range 1 (n + 1))
```

Poniższe przykłady zaadoptowane z [6].

```
let square = fn x → x * x
let fact = fn x →
  if x ⩽ 1 then 1 else x * fact (x - 1)

-- Sum of the results of applying a function to each element of a list.
let sigma = fn f l → case l of
  | [] → 0
  | x :: t → f x + sigma f t
let sumsquares = sigma (fn x → x * x)

-- Insertion sort.
let sort = fn l →
  let insert = fn e l → case l of
    | [] → [e]
    | x :: t → if e < x then e :: x :: t
               else x :: insert e t
  in
    case l of
      | [] → []
      | x :: t → insert x (sort t)

-- Silnia zaimplementowana za pomocą składania funkcji.
let compose = fn f g x → f (g x)
let power = fn f n →
  if n == 0 then fn x → x
  else compose f (power f (n - 1))


-- Obliczenia symboliczne.

type expr =
  | Num of int
  | Var of string
  | Let of string * expr * expr
  | Binop of string * expr * expr

let assoc = fn a l → case l of
  | [] → raise Not_found
  | (x, y) :: t → if x == a then y else assoc a t
let eval = fn env e → case e of
```

```
  | Num i → i
  | Var x → assoc x env
  | Let (x, e1, in_e2) →
      let val_x = eval env e1 in
      eval ((x, val_x) :: env) in_e2
  | Binop (op, e1, e2) →
      let v1 = eval env e1 in
      let v2 = eval env e2 in
      eval_op op v1 v2
and eval_op = fn op v1 v2 → case op of
  | "+" → v1 + v2
  | "-" → v1 - v2
  | "*" → v1 * v2
  | "/" → v1 / v2
  | _ → failwith ("Unknown operator: " ^ op)
```

Poniższe przykłady zaadoptowane z [7].

```
-- Last element of a list.
let last = fn l → case l of
  | [] → None
  | [x] → Some x
  | _ :: t → last t

-- Last two elements of a list.
let last_two = fn l → case l of
  | [] → None
  | [_] → None
  | [x, y] → Some (x, y)
  | _ :: t → last_two t

-- Nth element of a list.
let at = fn k l → case l of
  | [] → None
  | h :: t → if k == 1 then Some h
             else at (k - 1) t

-- Reverse a list.
let rev = fn l →
  let aux = fn acc l → case l of
    | [] → acc
    | h :: t → aux (h :: acc) t
  in
    aux [] list

-- Check if a list is a palindrome.
let is_palindrome = fn l →
  list_equal l (rev l)


-- Run-length encoding: encode and decode.

type 'a rle =
  | One of 'a
  | Many of int * 'a

let encode = fn l →
  let mktuple = fn cnt elem → case cnt of
```

```
    | 1 -> One elem
    | _ -> Many (cnt, elem)
  and aux = fn count acc l -> case l of
    | [] -> []
    | [x] -> (mktuple (count + 1) x) :: acc
    | a :: b :: t ->
      if a == b then aux (count + 1) acc (b :: t)
      else aux 0 ((mktuple (count + 1) a) :: acc) (b :: t)
  in
    rev (aux 0 [] l)

let decode = fn l ->
  let many = fn acc n x -> case n of
    | 0 -> acc
    | _ -> many (x :: acc) (n - 1) x
  and aux = fn acc l -> case l of
    | [] -> acc
    | One x :: t -> aux (x :: acc) t
    | Many (n, x) :: t -> aux (many acc n x) t
  in
    aux [] (rev l)
```

4. CENNIK

\+ = jest zaimplementowane

```
  Na 20 punktów
+ 01 (dwa typy)
+ 02 (arytmetyka, porównania)
+ 03 (if)
+ 04 (funkcje wieloargumentowe, rekurencja)
+ 05 (funkcje anonimowe i wyższego rzędu, częściowa aplikacja)
+ 06 (obsługa błędów wykonania)
+ 07 (statyczne wiązanie identyfikatorów)
  Listy:
+ 08 (z pattern matchingiem)
+ 09 (z empty, head, tail)
+ 10 (lukier)
  Na 25 punktów
+ 11 (listy dowolnego typu, zagnieżdżone i listy funkcji)
+ 12 (proste typy algebraiczne z jednopoziomowym pattern matchingiem)
+ 13 (statyczne typowanie)
  Na 30 punktów
+ 14 (ogólne polimorficzne i rekurencyjne typy algebraiczne)
+ 15 (zagnieżdżony pattern matching)
  Bonus
+ 16 (typy polimorficzne z algorytmem rekonstrukcji typów)
  17 (sprawdzenie kompletności pattern matchingu)

Razem: 34
```

5. BIBLIOGRAFIA

[1] Institut National de Recherche en Informatique et en Automatique (INRIA), „7.5 Operators", w *The OCaml User Manual*. [Online]. Dostępne na: https://v2.ocaml.org/manual/expr.html#ss%3Aexpr-operators

[2] „LBNF reference". [Online]. Dostępne na: https://bnfc.readthedocs.io/en/latest/lbnf.html

[3] „The BNF Converter". [Online]. Dostępne na: https://bnfc.digitalgrammars.com/

[4] Andreas Rossberg, „Standard ML Grammar". [Online]. Dostępne na: https://people.mpi-sws.org/~rossberg/sml.html

[5] Marcin Benke, „Język Latte". [Online]. Dostępne na: https://www.mimuw.edu.pl/~ben/Zajecia/Mrj2022/Latte/

[6] „Code Examples". [Online]. Dostępne na: https://v2.ocaml.org/learn/taste.html

[7] Victor Nicollet, „99 Problems (solved) in OCaml". [Online]. Dostępne na: https://v2.ocaml.org/learn/tutorials/99problems.html