

# Interpretor pentru un limbaj de programare funcțional

## Document de specificație a limbajului

### 1. Prezentare generală

Acest proiect constă în proiectarea și implementarea unui interpretor pentru un limbaj de programare funcțional, expresional, cu tipare statice, realizat în limbajul Lean. Limbajul este inspirat din familia ML.

Interpreterul suportă funcții de ordin superior, recursie, liste și potrivire de tipare (pattern matching). Implementarea este complet funcțională și este însorită de un set de teste care demonstrează corectitudinea comportamentului.

### 2. Sintaxă

#### 2.1. Elemente Lexicale

Limbajul utilizează următoarele categorii de tokeni:

- **Identifieri:** secvențe alfanumerice ce încep cu literă (x, fact, sum)
- **Literali:**
  - întregi: 0, 1, -5
  - booleeni: true, false
- **Operatori binari:**
  - aritmetici: +, -, \*
  - de comparație: <=, ==
- **Cuvinte cheie:**
  - if, then, else, let, in, letrec, fun, match, with

#### 2.2. Arborele de sintaxă abstractă (AST)

```
Expr ::= Int
      | Bool
      | Var
      | BinOp Expr Expr
      | If Expr Expr Expr
      | Let Var Expr Expr
      | Fun Var Type Expr
      | App Expr Expr
      | LetRec Var Var Type Type Expr Expr
      | Nil
      | Cons Expr Expr
      | Match Expr Expr Var Var Expr
```

## 2.3. Sintaxa concretă

```
e ::= n
    | true | false
    | x
    | e1 + e2 | e1 - e2 | e1 * e2
    | e1 <= e2 | e1 == e2
    | if e1 then e2 else e3
    | let x := e1 in e2
    | fun x : T => e
    | e1 e2
    | letrec f(x : T1) : T2 := e1 in e2
    | nil[T]
    | cons(e1, e2)
    | match e with | nil => e1 | cons h t => e2
```

## 3. Semantică

### 3.1. Model semantic

Interpreterul implementează:

- semantică operațională de tip big-step
- medii lexicale pentru variabile
- closure-uri pentru funcții

Evaluarea este deterministă și produce fie o valoare, fie o eroare semantică explicită.

### 3.2. Valori

Value ::= IntVal

```
| BoolVal
| Closure(parametru, corp, mediu)
| NilVal
| ConsVal(cap, coadă)
```

### 3.3. Reguli de evaluare (descriere informală)

Variabile: sunt evaluate prin căutare în mediul curent

Let: evaluatează expresia legată și extinde mediul

Funcții: produc closure-uri care capturează mediul

Aplicație: evaluatează funcția și argumentul, apoi corpul

LetRec: creează closure-uri recursive

Match: selectează ramura corespunzătoare structurii listei

## 4. Sistemul de tipuri

### 4.1. Tipuri

Type ::= Int

| Bool  
| Type -> Type  
| List Type

### 4.2 Mediul de tipuri

$\Gamma : \text{Variabilă} \rightarrow \text{Tip}$

### 4.3 Reguli de tipare (exemple)

#### Funcție

$\Gamma, x:T1 \vdash e : T2$

-----  
 $\Gamma \vdash \text{fun } x:T1 \Rightarrow e : T1 \rightarrow T2$

#### Aplicație

$\Gamma \vdash e1 : T1 \rightarrow T2 \quad \Gamma \vdash e2 : T1$

-----  
 $\Gamma \vdash e1\ e2 : T2$

#### LetRec

Funcțiile recursive sunt introduse în mediul de tipuri cu tipul  $T1 \rightarrow T2$  înainte de verificarea corpului.

### 4.4 Verificarea tipurilor

- Sistem de tipuri static
- Fără inferență de tipuri
- Erorile de tip sunt detectate înainte de evaluare

## 5. Decizii de proiectare

#### • Limbaj funcțional pur

Eliminarea mutabilității simplifică semantica și evidențiază concepte precum closures și scope lexical.

#### • Recursie în loc de bucle imperative

Repetarea este realizată prin letrec, fiind echivalentă expresiv cu buclele clasice.

- **Separarea clară a componentelor**  
Parserul, semantica și sistemul de tipuri sunt implementate separat, pentru claritate.
- **Control explicit al aplicației funcționale**  
Aplicația este oprită în prezența operatorilor binari pentru a evita ambiguități de parsare.
- **Parser implementat manual**  
Demonstrează înțelegerea completă a procesului de analiză sintactică.

## Documentație pentru utilizator

Această secțiune descrie modul în care poate fi utilizat limbajul implementat, prin explicații pas cu pas și exemple concrete de cod.

### 1. Rularea programelor

Evaluarea unui program se face folosind funcția:

```
#eval runString fuelTest "program"
```

unde "program" este un sir de caractere ce conține codul scris în limbajul definit.  
Interpreterul:

1. tokenizează programul,
2. îl parsează într-un AST,
3. verifică tipurile,
4. evaluează expresia.

Dacă programul este corect, se obține o valoare. În caz contrar, este afișat un mesaj de eroare (de parsare sau de tip).

### 2. Expresii aritmetice

Limbajul suportă operații aritmetice pe numere întregi.

### Operatori disponibili

- + – adunare
- - – scădere
- \* – înmulțire

Operatorii respectă **precedența standard** (\* are prioritate față de + și -).

### Exemple

- $2 + 3 * 4$

Rezultat: 14

- $(2 + 3) * 4$

Rezultat: 20

### 3. Expresii booleene și comparații

Sunt disponibile următoarele operații de comparație:

- $\leq$  – mai mic sau egal
- $\equiv$  – egalitate

Rezultatul este de tip boolean (true sau false).

### Exemple

- $1 \leq 2$

Rezultat: true

- $3 \equiv 4$

Rezultat: false

### 4. Instrucțiuni condiționale (if)

Instrucțiunea condițională are forma:

if condiție then expresie1 else expresie2

Condiția trebuie să fie de tip bool.

### Exemplu

- if  $1 \leq 2$  then 10 else 20

Rezultat: 10

## **5. Variabile și legări (let)**

Variabilele pot fi definite folosind construcția let.

### **Sintaxă**

let x := expresie1 in expresie2

Variabila x este vizibilă doar în expresie2.

### **Exemplu simplu**

- let x := 5 in x + 1

Rezultat: 6

### **Scope lexical (domeniu de vizibilitate)**

{ let x := 10 in

    let x := 20 in

        x }

Rezultat: 20

Variabila cea mai recent definită o ascunde pe cea anterioară.

## **6. Funcții**

Funcțiile sunt expresii de primă clasă și pot fi definite folosind fun.

### **Sintaxă**

fun x : tip => expresie

### **Exemplu**

- (fun x : int => x + 1) 5

Rezultat: 6

## **7. Funcții de ordin superior**

Funcțiile pot primi alte funcții ca argumente sau pot returna funcții.

## **Exemplu**

```
{ let apply :=  
    fun f : int -> int =>  
        fun x : int => f x  
    in  
        apply (fun y : int => y * 2) 10 }
```

Rezultat: 20

## **8. Funcții recursive (letrec)**

Funcțiile recursive sunt definite folosind letrec.

### **Sintaxă**

letrec f(x : T1) : T2 := expresie in expresie\_finală

#### **Exemplu: factorial**

```
{ letrec fact (n : int) : int :=  
    if n <= 0 then 1 else n * fact (n - 1)  
in fact 5 }
```

Rezultat: 120

## **9. Liste**

Limbajul suportă liste omogene (toate elementele au același tip).

### **Constructori**

- nil[T] – lista vidă de tip T
- cons(h, t) – adaugă elementul h în fața listei t

## **Exemplu**

cons(1, cons(2, cons(3, nil[int])))

### **3.10. Potrivire de tipare (match)**

Listele pot fi analizate folosind match.

#### **Sintaxă**

match expresie with

| nil => expresie1

| cons h t => expresie2

#### **Exemplu**

match cons(1, cons(2, nil[int])) with

| nil => 0

| cons h t => h

Rezultat: 1

### **3.11 Erori de tip (exemple)**

Interpreterul detectează erorile de tip înainte de evaluare.

#### **Aplicarea unei valori non-funcție**

(1 + 2) 3

Mesaj: TYPE ERROR : expected function

#### **Tipuri incompatibile**

1 true

Mesaj: TYPE ERROR : expected int, got bool

### **3.12. Observații finale**

Toate funcțiile trebuie să aibă tipuri explicit specificate.

Limbajul este strict tipat (nu există conversii implicate).

Aplicația funcțională nu este permisă peste operatori binari, pentru a evita ambiguități de parsare.

Mesajele de eroare indică exact cauza problemei (parsare sau tip).

## **Concluzie**

Limbajul oferă un set coerent și expresiv de funcționalități pentru programare funcțională: funcții, recursie, liste și potrivire de tipare. Documentația de față permite utilizarea limbajului fără cunoașterea implementării interne și demonstrează clar comportamentul fiecărei construcții.