Proiectarea Semanticii Limbajului JavaScript-PLP

Documentatie Tehnica

January 11, 2026

Contents

1 Introducere

Acest document descrie semantica operationala a unui interpretor pentru un subset al limbajului JavaScript. Interpretorul este implementat in OCaml si suporta:

- Tipuri de date: Int, Float, Bool, Char, String, Undefined
- Expresii aritmetice, logice si de comparatie
- Instructiuni: declaratii, atribuiri, if-else, while, return
- Functii cu suport pentru recursivitate
- Domenii de vizibilitate (block scoping) cu shadowing

2 Modelul Semanticii Operationale

Semantica limbajului este definita utilizand **Semantica Operationala Big-Step**. Deoarece in acest limbaj atribuirea este o expresie, evaluarea oricarei expresii poate modifica starea programului.

2.1 Starea Programului (Environment)

Starea programului, notata cu σ, este o functie care mapeaza identificatorii la valorile lor:

$$\sigma \in \text{Env} = \text{Var} \to \text{Value} \cup \{\bot\}$$

unde Value = {VInt, VFloat, VBool, VChar, VString, VUndefined, VFunc}.

2.2 Valorile Functiilor

O valoare de tip functie contine:

$$\text{VFunc}(\text{params}, \text{body}, \sigma_{closure})$$

unde:

- `params` - lista parametrilor formali
- `body` - corpul functiei (instructiune)
- $\sigma_{closure}$ - environment-ul capturat la momentul definirii (closure)

3 Semantica Expresiilor

Evaluarea unei expresii e in starea σ produce o valoare v si o stare noua σ' (datorita posibilelor atribuiri):

$$\langle e, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$$

3.1 Reguli pentru Expresii

3.1.1 Constante si Variabile

- **Constante:** $\langle c, \sigma \rangle \Downarrow \langle c, \sigma \rangle$
- **Variabile:** $\dfrac{\sigma(x) = v}{\langle x, \sigma \rangle \Downarrow \langle v, \sigma \rangle}$

3.1.2 Atribuire

Atribuirea este o expresie care returneaza valoarea atribuita:

$$\frac{\langle e, \sigma \rangle \Downarrow \langle v, \sigma' \rangle \quad x \in \text{dom}(\sigma')}{\langle x = e, \sigma \rangle \Downarrow \langle v, \sigma'[x \leftarrow v] \rangle}$$

3.1.3 Operatii Binare

$$\frac{\langle e_1, \sigma \rangle \Downarrow \langle v_1, \sigma'' \rangle \quad \langle e_2, \sigma'' \rangle \Downarrow \langle v_2, \sigma' \rangle \quad v = v_1 \oplus v_2}{\langle e_1 \oplus e_2, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}$$

Nota: e_2 se evalueaza in starea σ'' produsa de e_1 (evaluare de la stanga la dreapta).

3.1.4 Operatii Unare

$$\frac{\langle e, \sigma \rangle \Downarrow \langle v_1, \sigma' \rangle \quad v = \ominus v_1}{\langle \ominus e, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}$$

4 Semantica Functiilor

4.1 Declaratia Functiilor

Declaratia unei functii adauga o valoare de tip `VFunc` in environment:

$$\frac{f_{val} = \text{VFunc}(\text{params}, \text{body}, \sigma)}{\langle \text{function } f(\text{params})\{\text{body}\}, \sigma \rangle \rightarrow \sigma[f \leftarrow f_{val}]}$$

4.2 Apelul Functiilor

Apelul unei functii implica:

1. Evaluarea argumentelor (de la stanga la dreapta)
2. Crearea unui environment local cu parametrii legati la argumente
3. Executia corpului functiei in environment-ul local
4. Returnarea valorii (sau `undefined` daca nu exista `return`)

$$\frac{\begin{array}{c}\sigma(f) = \text{VFunc}(\text{params}, \text{body}, \sigma_c) \\ \langle \text{args}, \sigma \rangle \Downarrow \langle \text{vals}, \sigma' \rangle \\ \sigma_{call} = \sigma_c[f \leftarrow \sigma(f)][\text{params} \leftarrow \text{vals}] \\ \langle \text{body}, \sigma_{call} \rangle \rightarrow Exit(v)\end{array}}{\langle f(\text{args}), \sigma \rangle \Downarrow \langle v, \sigma' \rangle}$$

4.3 Recursivitate

Pentru a suporta recursivitatea, functia este adaugata in propriul environment de apel:

$$\sigma_{call} = \sigma_{closure}[f \leftarrow \text{VFunc}(\text{params}, \text{body}, \sigma_{closure})]$$

Astfel, in corpul functiei, apelurile recursive gasesc functia in environment.

4.4 Exemplu: Factorial Recursiv

```
function factorial(n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
let result = factorial(5);  // result = 120
```

5 Semantica Instructiunilor

Instructiunile transforma starea σ. Introducem o stare speciala $Exit(v)$ pentru a gestiona instructiunea `return`.

5.1 Reguli de Baza

- **Skip:** $\langle ;, \sigma \rangle \to \sigma$
- **Declaratie cu initializare:**

$$\frac{\langle e, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \text{let } x = e, \sigma \rangle \to \sigma'[x \leftarrow v]}$$

- **Declaratie fara initializare:**

$$\langle \text{let } x, \sigma \rangle \to \sigma[x \leftarrow \text{undefined}]$$

- **Expresie ca instructiune:**

$$\frac{\langle e, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle e; , \sigma \rangle \to \sigma'}$$

5.2 Structuri de Control

5.2.1 Conditionala (If-Else)

$$\frac{\langle e, \sigma \rangle \Downarrow \langle \text{true}, \sigma'' \rangle \quad \langle s_1, \sigma'' \rangle \to \sigma'}{\langle \text{if } (e)\ s_1 \text{ else } s_2, \sigma \rangle \to \sigma'}$$

$$\frac{\langle e, \sigma \rangle \Downarrow \langle \text{false}, \sigma'' \rangle \quad \langle s_2, \sigma'' \rangle \to \sigma'}{\langle \text{if } (e)\ s_1 \text{ else } s_2, \sigma \rangle \to \sigma'}$$

5.2.2 Bucla While

$$\frac{\langle e, \sigma\rangle \Downarrow \langle \text{false}, \sigma'\rangle}{\langle \text{while } (e)\ s, \sigma\rangle \to \sigma'}$$

$$\frac{\langle e, \sigma\rangle \Downarrow \langle \text{true}, \sigma''\rangle \quad \langle s, \sigma''\rangle \to \sigma''' \quad \langle \text{while } (e)\ s, \sigma'''\rangle \to \sigma'}{\langle \text{while } (e)\ s, \sigma\rangle \to \sigma'}$$

5.2.3 Return

Instructiunea `return` opreste executia si propaga valoarea:

$$\frac{\langle e, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}{\langle \text{return } e, \sigma\rangle \to Exit(v)}$$

6 Domenii de Vizibilitate (Scoping)

6.1 Block Scoping

Variabilele declarate intr-un bloc { } sunt vizibile doar in acel bloc. La iesirea din bloc, variabilele locale sunt eliminate:

$$\frac{\langle \text{stmts}, \sigma\rangle \to \sigma_{final}}{\langle \{\text{stmts}\}, \sigma\rangle \to \sigma'}$$

unde starea rezultata σ' pastreaza doar variabilele pre-existente:

$$\sigma'(x) = \begin{cases} \sigma_{final}(x) & \text{daca } x \in \text{dom}(\sigma) \\ \bot & \text{daca } x \notin \text{dom}(\sigma) \end{cases}$$

6.2 Shadowing

O variabila declarata intr-un bloc poate "umbri" o variabila cu acelasi nume din scope-ul exterior. La iesirea din bloc, valoarea originala este restaurata.

```
let x = 100;              // x global
{
    let x = 999;          // x local (shadowing)
    // aici x = 999
}
// aici x = 100 (restaurat)
```

6.3 Modificari vs Redeclarari

- **Atribuire** ($x = v$): Modifica valoarea variabilei existente. Modificarea persista dupa iesirea din bloc.
- **Declaratie** (let $x = v$): Creaza o variabila noua locala. Variabila este eliminata la iesirea din bloc.

6.4 Exemplu Complet

```
let global = 100;
let modificata = 10;
let shadowed = "original";

{
    let local = 50;
    modificata = modificata + local;  // Modificare
    let shadowed = "in bloc";         // Shadowing
}
// Rezultat:
// global = 100
// modificata = 60  (modificarea persista)
// shadowed = "original"  (valoarea restaurata)
// local nu exista (eliminata)
```

7 Sistemul de Tipuri

Limbajul utilizeaza **tipizare dinamica puternica** (strong dynamic typing). Operatiile intre tipuri incompatibile genereaza erori la runtime.

7.1 Tipuri Suportate

- `VInt` - numere intregi
- `VFloat` - numere reale
- `VBool` - valori booleene
- `VChar` - caractere
- `VString` - siruri de caractere
- `VUndefined` - valoare nedefinita
- `VFunc` - functii (first-class values)

7.2 Coercitie Limitata

Limbajul permite cateva conversii implicite:

- `Int + Float` → `Float`
- `String + Int` → `String` (concatenare)

8 Tratarea Erorilor

Interpretorul genereaza urmatoarele tipuri de erori:

- `RuntimeError` - erori generale (impartire la zero, etc.)
- `UndefinedVariable` - variabila nedeclarata
- `TypeError` - operatie pe tipuri incompatibile

9 Sintaxa Limbajului (BNF)

```
<program>      ::= <stmt>*
<stmt>         ::= 'let' <id> '=' <expr> ';'
                 | 'let' <id> ';'
                 | <expr> ';'
                 | 'if' '(' <expr> ')' <stmt> 'else' <stmt>
                 | 'while' '(' <expr> ')' <stmt>
                 | '{' <stmt>* '}'
                 | 'function' <id> '(' <params> ')' <stmt>
                 | 'return' <expr> ';'
<params>       ::= <id> (',' <id>)* |
<expr>         ::= <int> | <float> | <bool> | <char> | <string> | <id>
                 | <id> '=' <expr>
                 | <expr> <binop> <expr>
                 | <unop> <expr>
                 | <id> '(' <args> ')'
                 | 'function' '(' <params> ')' <stmt>
                 | '(' <expr> ')'
<args>         ::= <expr> (',' <expr>)* |
<binop>        ::= '+' | '-' | '*' | '/' | '%' | '==' | '!=' | '<' | '<=' | '>' | '>=
<unop>         ::= '-' | '!'
<id>           ::= [a-zA-Z_][a-zA-Z0-9_]*
<int>          ::= [0-9]+
<float>        ::= [0-9]+'.'[0-9]+
<bool>         ::= 'true' | 'false'
<char>         ::= '\'' . '\''
<string>       ::= '"' .* '"'
```

10 Decizii de Proiectare

- **Limbaj de implementare:** OCaml, pentru ușurința manipulării structurii de date și pattern matching.
- **Atribuirea este expresie:** Permite scrierea de expresii complexe și assignment chaining.
- **Block scoping:** Domeniile de vizibilitate sunt ca în JavaScript (ES6+), cu shadowing și eliminarea variabilelor locale la ieșirea din bloc.
- **Funcții ca valori de primă clasă:** Suportă closures și recursivitate.
- **Tipizare dinamică:** Verificarea tipurilor se face la execuție, nu la compilare.
- **Erori explicite:** Orice acces la variabile nedeclarate sau operații pe tipuri incompatibile opresc execuția cu mesaj clar.

11 Exemple de Utilizare

11.1 Expresii și Control

```
let x = 10;
let y = 20;
let suma = x + y * 2;
if (suma > 30) {
    suma = suma - 5;
} else {
    suma = suma + 5;
}
```

11.2 Funcții și Recursivitate

```
function factorial(n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
let f5 = factorial(5); // f5 = 120
```

11.3 Domenii de Vizibilitate

```
let a = 1;
{
    let a = 2;
    // aici a = 2
}
// aici a = 1
```

11.4 Closure

```
function make_adder(x) {
    return function(y) { return x + y; };
}
let add5 = make_adder(5);
let z = add5(3); // z = 8
```

12 Testare și Exemplu de Rulare

Testele pentru interpretor se găsesc în fișierul `teste_de_incercat`. Acesta conține peste 20 de programe care acoperă toate funcționalitățile cerute:

- Literali și tipuri de bază
- Operații aritmetice și logice
- Control flow: if, while
- Funcții, recursivitate, closure
- Domenii de vizibilitate și shadowing
- Return și blocuri imbricate
- Erori de tip și variabile nedeclarate

Pentru a rula un test, copiați programul dorit în `test.js` și executați:
`./run.sh`
`./interpretor`
Rezultatul va fi afișat în consolă, inclusiv starea finală a variabilelor sau valoarea returnată.
Exemple de output se găsesc în secțiunea de exemple și în fișierul de teste.