Proiect pentru obținerea atestării profesionale în informatică

# Octarou: Interpretor pentru limbajul de programare `CHIP-8`

Nicolas-Ștefan Bratoveanu
Prof. coordonator Mihaela Stan

Colegiul Național „Vasile Alecsandri” Galați
2023-2024

# Cuprins

# 1 Introducere

## 1.1 Contextualizare

*Octarou* este un interpretor pentru limbajul de programare `CHIP-8`. Acesta este un limbaj de programare interpretat, dezvoltat de către Joseph Weisbecker în 1977 pentru sisteme bazate pe microprocesorul RCA 1802. A fost inițial utilizată pe computerele COSMAC VIP și Telmac 1800. Limbajul a fost creat cu scopul de a permite dezvoltarea mult mai ușoară a jocurilor video pe aceste platforme, permițând utilizarea unor instrucțiuni hexazecimale, în locul instrucțiunilor native [6]. Spre sfârșitul anilor '70, în spatele acestui limbaj se formase o comunitate activă de dezvoltatori și utilizatori, care a luat naștere odată cu buletinul informativ *VIPer* al revistei *ARESCO*, ale cărei prime ediții au dezvăluit codul pentru interpretorul original de `CHIP-8`.

CHIP-8 s-a răspândit pe alte platforme, precum computerele australiene DREAM 6800, ETI-660 și MicroBee, computerul finlandez menționat mai devreme, Telmac 1800 și computerul canadian ACE VDU [6].

Ulterior, au apărut interpretoare derivate și extensii la limbajul original, folosite, spre exemplu, pe calculatoare grafice (`CHIP-48`, `SUPER-CHIP` pentru calculatoarele HP-48 [6]). Aceste dispozitive aparțineau perioadei de după anii '80 și aveau, de regulă, mult mai multă putere de procesare decât microcomputerele din deceniul precedent, cum ar fi COSMAC VIP-ul.

## 1.2 Motivația alegerii temei

La nivel de suprafață, alegerea acestei teme poate părea (cel puțin) dubioasă. Cu toate că este, într-adevăr, o temă destul de obscură, consider că este extrem de valoroasă. Astfel de proiecte au valoare din punct de vedere al istoriei calculatoarelor, precum și în cadrul fenomenului de „retro-computing". Mai mult decât atât, CHIP-8, în mod specific, are o valoare educațională deosebită, întrucât implementarea unui astfel de sistem este adesea recomandată ca prim pas în lumea dezvoltării de emulatoare [6].

# 2 Mediul de dezvoltare

## 2.1 Rust

Am ales Rust ca limbaj principal de dezvoltare pentru acest proiect. Rust este un limbaj de programare general, multi-paradigmă, care pune accent pe siguranță, eficiență și paralelism. Este utilizat în numeroase situații, de la programarea sistemelor, la dezvoltarea de aplicații grafice, dezvoltarea web sau a sistemelor integrate.

Performanța și eficiența se datorează absenței unui runtime și a unui *garbage collector*. Siguranța memoriei este asigurată de către sistemul de tipuri bogat și de modelul de *ownership*, care permit eliminarea a numeroase tipuri de buguri în stagiul de compilare [8]. Modelul de *ownership* impune niște reguli privitoare la deținători [5, Cap. 4.1]:

- Orice valoare are un deținător (*owner*).
- Nu poate exista decât un deținător la un moment dat.
- Când deținătorul iese din scope, valoarea este eliberată.

și la utilizarea referințelor [5, Cap. 4.2]:

- Referințele sunt mereu valide.
- La un moment dat, pot exista fie mai multe referințe imuabile, fie o singură referință muabilă, dar niciodată ambele.

```rust
#[derive(Debug)]
struct NonZeroU8(u8);

impl NonZeroU8 {
    fn new(value: u8) -> Option<Self> {
        match value {
            0 => None,
            _ => Some(Self(value))
        }
    }
}
```

Listing 1: Exemplu de structură cu invariantă simplă

Sistemul de tipuri este extrem de puternic și permite modelarea stării unei aplicații în moduri idiomatice, prin utilizarea tipurilor pentru a impune invariantele. Spre exemplu, considerăm tipul de date din Listing 1. Invarianta acestuia este că valoarea pe care o memorează este diferită de 0. Acest lucru este reprezentat direct în sistemul de tipuri prin utilizarea `Option<T>` (echivalent cu `Maybe a` din Haskell sau `std::optional<T>` din C++17), un tip polimorfic utilizat pentru a încapsula o valoare sau *absența acesteia*. Dacă valoarea pe care o primește constructorul `new` este nulă, acesta va întoarce valoarea `None`, pe care apelatorul este forțat să o gestioneze.

Dacă se încearcă utilizarea valorii de tip `Option<u8>` ca atare, se obține o eroare de compilare.

```rust
let a = NonZeroU8::new(0);
println!("{:?}", a + 2);
```

```
error[E0369]: cannot add `{integer}` to `Option<NonZeroU8>`
  --> src/main.rs:15:24
   |
15 |     println!("{:?}", a + 2);
   |                       - ^ - {integer}
   |                       |
   |                       Option<NonZeroU8>
```

„Capturarea” erorilor în stagiul de compilare este una dintre super-puterile acestui limbaj.

### 2.1.1 Librăriile `eframe` și `egui`

**Egui** este o librărie simplă, rapidă și portabilă de tip *immediate mode GUI* scrisă în Rust pur [3], iar **eframe** este o librărie utilizată alături de egui care pune la dispoziție interacțiunea dintre aceasta și platforma specifică pe care o țintește aplicația (în cazul acesta, Windows, macOS, Linux, BSD și, parțial, WebAssembly). Spre deosebire de alte librării de UI de tip *retained mode*, folsind egui, aplicația redesenează procedural UI-ul în fiecare cadru. Am ales acest mediu grafic pentru simplitate și, totodată, datorită faptului că aceste librării sunt scrise în întregime în Rust și beneficiază, prin urmare de avantajele de securitate, corectitudine și performanță asociate.

## 2.2 Nix

Nix este un gestionar de pachete pentru sisteme Unix-like bazat pe un *model pur funcțional de implementare a software-ului* [2]. Acesta gestionează problema implementării și

```
packages.default = craneLib.buildPackage (commonArgs
  // {
    inherit cargoArtifacts;

    buildInputs = with pkgs;
      [pkg-config]
      ++ lib.optional (stdenv.isLinux) alsa-lib;

    LD_LIBRARY_PATH = libPath;
  });
```

Listing 2: Derivația procesului de compilare pe platforma nativă

```
toolchainWindows = with fenix.packages.${system};
  combine [
    stable.rustc
    stable.cargo
    targets.x86_64-pc-windows-gnu.stable.rust-std
  ];
```

Listing 3: Toolchainul utilizat pentru cross-compilare pentru Windows

distribuirii pachetelor de software prin instalarea lor într-o locație centralizată, în directoare unice, discriminate printr-un hash criptografic. Astfel, elimină problemele asociate lipsei de dependențe și permite coexistența mai multor versiuni ale aceluiași pachet (aici, versiune nu se referă doar la numărul de versiune semantică, ci include și parametrii procesului de compilare a pachetului).

*Octarou* se bazează pe un Nix flake atât în procesul de dezvoltare cât și în cel de compilare și distribuție. Fișierul `flake.nix` conține o descriere declarativă (*derivație*) a procesului de compilare al aplicației (Listing 2), dar și a shellului utilizat pentru dezvoltarea acesteia. Pentru aceasta am utilizat librăria *Crane*[1], care facilitează descărcarea dependințelor (versiuni fixe, declarate în `flake.lock`) și compilare incrementală și compozabilă.

Pachetele Nix sunt definite cu ajutorul unui limbaj funcțional cu evaluare leneșă concepută special pentru gestionarea pachetelor. Dependințele sunt urmărite folosind un format intermediar (*derivațiile*), care se află, alături de restul pachetelor, într-o locație centralizată numită *Nix store* (de obicei, la `/nix/store`). Derivațiile și specificațiile derivațiilor din Nix store sunt eliminate automat prin *garbage collection*. Acest model funcțional garantează că orice actualizare a pachetelor este *atomică* și permite rollback-uri în timp constant $\mathcal{O}(1)$ [2].

Repozitoriul principal în care se poate găsi întreaga colecție de pachete ce pot fi distribuite prin Nix se numește `nixpkgs`. Acesta găzduiește și codul-sursă pentru NixOS, o distribuție de Linux bazată pe gestionarul de pachete Nix, care este întru totul configurabilă prin limbajul Nix.

### 2.2.1 Cross-compilare

Pachetele pentru `x86_64-pc-windows-gnu` sunt compilate prin Nix și Crane, folosind facilitățile de cross-compilare (Listing 3) din `nixpkgs`. Procesul acesta este declanșat automat printr-un pipeline de CI pentru fiecare versiune semantică a proiectului. Un mecanism similar este utilizat pentru compilarea pentru `wasm32-unknown-unknown`, arhitectura-țintă pentru platforma web[2].

[1] O librărie Nix pentru compilarea proiectelor Rust prin Cargo. (https://crane.dev/)

[2] WebAssembly încă nu susține toate API-urile utilizate în logica interpretorului (`std::time::Instant`), motiv pentru care, pe această platformă, aplicația se comportă în moduri non-ideale.

| **Componentă** | **Mărime** | **Utilizare** |
|---|---|---|
| Memorie | 4 KB, adrese pe 16 biți | Memorie volatilă |
| Display | 64x32 pixeli | Grafică |
| Stivă | practic nelimitată, adrese pe 16 biți | Structură de tip LIFO; utilizată pentru apelarea subprogramelor |
| Delay | 8 biți | Decrementat cu o frecvență de 60 Hz; utilizat pentru calculul timpilor în aplicații |
| Sound | 8 biți | Decrementat la fel ca Delay; utilizat pentru gestionarea sunetelor |
| Index | 16 biți | Registru utilizat pentru a indexa memoria (pentru sprite-uri, font etc.) |
| Program Counter | 16 biți | Registru care indică adresa de memorie a instrucțiunii ce urmează să fie executată |
| Variables | 16 registre de 8 biți | Sunt numerotate de la V0 la VF (hexazecimal); folosite ca variabile; VF este folosit ca *flag register* |

Tabela 1: Componentele interpretorului `CHIP-8` [6]

# 3 Funcționarea `CHIP-8`

Interpretorul `CHIP-8` operează pe componentele enumerate în Tabela 1, citind instrucțiuni din memorie și executându-le, modificând memoria și registrele corespunzător.

## 3.1 Instrucțiunile

Instrucțiunile `CHIP-8` sunt reprezentate prin numere de 16 biți, citite din memorie, de la adresa indicată de program counter. Simbolic, acestea se reprezintă prin 4 cifre hexazecimale. Primii 4 biți (primul *nibble*) discriminează categoriile de instrucțiuni.

Majoritatea instrucțiunilor au una din formele următoare. Cu toate că valorile din cadrul codului instrucțiunii sunt mai mici de 8 sau 16 biți, ele sunt tratate ca valori de 8 respectiv 16 biți. În implementația aceasta, valorile de 16 biți sunt tratate ca `usize`, deoarece sunt folosite pentru a indexa tablouri.

- `00EE` – formă literală; o valoare fixă
- `0NNN` – formă parametrică; `NNN` reprezintă o valoare pe 16 biți (de obicei o adresă de memorie)
- `0XNN` – formă parametrică; `X` reprezintă o valoare pe 8 biți (de obicei indexul unui registru de variabile), iar `NN`, o valoare pe 16 biți (de obicei o adresă de memorie)
- `0XYN` – formă parametrică; `X` și `Y` reprezintă indecși pentru registrele de variabile, iar `N`, o valoare pe 8 biți

```rust
fn xyn(opcode: u16) -> (usize, usize, u8) {
    (
        ((opcode & 0x0F00) >> 8) as usize,
        ((opcode & 0x00F0) >> 4) as usize,
        (opcode & 0x000F) as u8,
    )
}

fn xnn(opcode: u16) -> (usize, u8) {
    (((opcode & 0x0F00) >> 8) as usize, (opcode & 0x00FF) as u8)
}

fn nnn(opcode: u16) -> usize {
    (opcode & 0x0FFF) as usize
}
```

Listing 4: Funcțiile utilizate pentru parsarea codurilor de instrucțiuni

## 3.2 Memoria

Memoria are în total 4 KB, adică $2^{12} = 4096$ de octeți. Acest spațiu este adresabil cu adrese de 12 biți, însă, în practică, se folosesc adrese de 16 biți. Spre exemplu, registrul *index* (`I`) reține o adresă de 16 biți care indică locația din memorie a sprite-urilor ce trebuie desenate pe ecran.

Pe COSMAC VIP, zona de memorie dintre 0 și 512 biți era ocupată de codul interpretorului (din pricina limitațiilor tehnice), însă în cazul interpretoarelor moderne, acest lucru nu este necesar; memoria aceea este utilizată, în schimb, pentru stocarea fontului, care se pune, de regulă, în zona 80–160 biți (Figura 1).

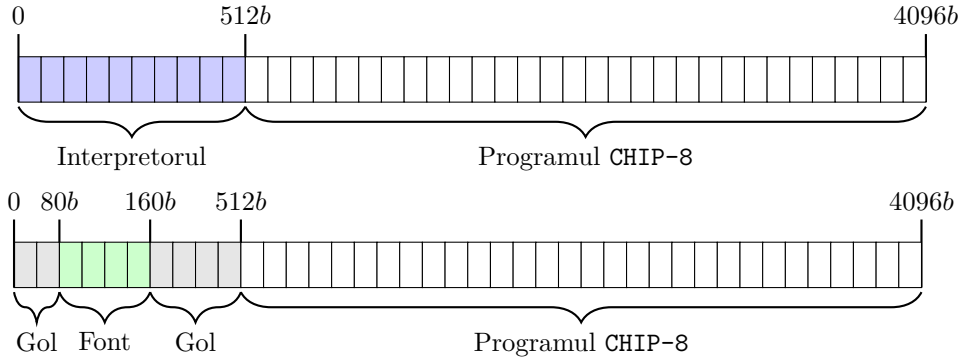


Figura 1: Layoutul de memorie `CHIP-8` (NU la scară)

Toată memoria este modificabilă, iar programele interpretate se pot și se vor modifica pe ele însele [6].

### 3.2.1 `ANNN`, `FX1E` și `FX29`

Toate aceste instrucțiuni manipulează registrul `I`. `ANNN` (*SetIndex*) pur și simplu setează `I` ← `NNN`, iar `FX1E` (*AddIndex*) adaugă `VX` la `I` (`I` ← `I` + `NNN`). Instrucțiunea `FX29` se numește *SetIndexFont* și setează `I` la adresa din memorie a caracterului reprezentat de ultimii 4 biți (ultimul *nibble*) ai valorii din `VX`.

### 3.2.2 `FX55` și `FX65`

Aceste instrucțiuni sunt folosite pentru stocarea și citirea din memorie a valorilor din registrele variabile. *StoreMemory*, `FX55`, scrie succesiv în memorie, începând de la adresa `I`, valorile din registrele `V0` până la `VX`. *LoadMemory*, `FX65` face opusul: citește din memorie, începând de la adresa `I`, valorile din registrele `V0`–`VX`.

În interpretorul de pe COSMAC VIP, această instrucțiune incrementa valoarea din `I`, astfel încât la finalul execuției instrucțiunii aceasta memora valoarea `I+X+1`. Interpretoarele moderne (de la `CHIP-48` și `SUPER-CHIP` înainte) nu fac acest lucru; valoarea indexului rămâne neschimbată [6][9].

## 3.3 Ecranul, sprite-urile și fontul

Ecranul este actualizat cu o rată de 60 Hz. Fiecare pixel este fie pornit fie oprit.

Un sprite este o secvență de octeți cu lungimea $1 \leq n \leq 15$. Fiecare octet reprezintă un „rând", iar fiecare bit din octet reprezintă un pixel. Instrucțiunea care desenează sprite-uri pe ecran este `DXYN` și va fi detaliată mai târziu, însă, pe scurt, ea efectuează o operațiune analoagă disjuncției logice exclusive (*exclusive or*) între fiecare pixel al sprite-ului și cel corespunzător de pe ecran [6].

Fontul este alcătuit din 16 sprite-uri de 4x5 pixeli care reprezintă cifrele hexazecimale 0–F. Caracterele sunt desenate pe ecran la fel ca sprite-urile obișnuite [6].

### 3.3.1 `00E0` și `DXYN`

Instrucțiunea `00E0` (*Clear*) resetează ecranul, adică setează toți pixelii la valoarea 0.

Instrucțiunea `DXYN` (*Draw*) este mai complexă. Aici, `X` și `Y` reprezintă registrele variabile care conțin coordonatele de pe ecran la care trebuie desenat sprite-ul indicat de registrul `I` (*index*), iar `N` reprezintă înălțimea sprite-ului care trebuie desenat. Cu alte cuvinte, trebuie desenat pe ecran sprite-ul din zona de memorie `I..I+N`, la coordonatele VX, respectiv, VY [3]. Coordonatele din VX și VY sunt reduse modulo 64, respectiv, 32, adică se „învârt" după marginile ecranului. În schimb, dacă sprite-ul depășește marginile ecranului, acesta este pur și simplu tăiat.

**Algoritm 1:** Desenarea unui sprite

```
//Coordonatele inițiale se „învârt" în jurul ecranului
x ← VX mod 64;
y ← VY mod 32;
VF ← 0;
pentru dy ← 0, N − 1 execută
    pentru dx ← 0, 7 execută
        //Sprite-ul este tăiat la marginile ecranului
        dacă y + dy < 32 și x + dx < 64 atunci
            //D este un tablou bidimensional ce reprezintă ecranul
            //M este un tablou ce reprezintă memoria
            b ← al dx-lea bit din M[I + dy];
            VF ← b ∧ D[y + dy][x + dx];
            D[y + dy][x + dx] ← b ⊕ D[y + dy][x + dx];
        sfârșit
    sfârșit
sfârșit
```

[3] În implementarea acestei instrucțiuni, valoarea registrului `I` nu este modificată în niciun fel.

Pentru fiecare pixel din sprite, acesta modifică pixelul de pe ecran cu care se suprapune prin operația de disjuncție exclusivă. Dacă vreun pixel de pe ecran a fost dezactivat (setat la 0) de această instrucțiune, atunci se setează VF la 1, altfel, VF este setat la 0 [6]. Algoritmul 1 reprezintă în linii mari, în pseudocod, procesul de desenare a unui sprite pe ecran.

## 3.4 Controlul fluxului și apelarea subprogramelor

Controlul fluxului se realizează printr-o serie de instrucțiuni de tip *jump* sau *skip*.

### 3.4.1 `1NNN` și `BNNN/BXNN`

Instrucțiunile `1NNN` (*Jump*) și `BNNN` (*JumpOffset*) sunt utilizate pentru a seta program counterul la o adresă de memorie specifică. *Jump* setează `PC` la adresa `NNN`. *JumpOffset* este mai ambiguă [6]. În interpretorul original de `CHIP-8`, această instrucțiune sărea la adresa `NNN` + `V0`. În `CHIP-48` și `SUPER-CHIP`, ea funcționează ca `BXNN`, sărind la adresa `XNN` + `VX`.

### 3.4.2 `3XNN, 4XNN, 5XY0` și `9XY0`

Aceste instrucțiuni sar peste următoarea instrucțiune, adică incrementează `PC` cu 2, astfel:

- `3XNN` va sări peste o instrucțiune dacă `VX` = `NN`.
- `4XNN` va sări peste o instrucțiune dacă `VX` ≠ `NN`.
- `5XY0` va sări peste o instrucțiune dacă `VX` = `VY`.
- `9XY0` va sări peste o instrucțiune dacă `VX` ≠ `VY`.

Combinându-le cu instrucțiunile *jump*, se poate obține un mecanism de control al fluxului similar cu `if`-ul din limbajele de programare moderne.

### 3.4.3 `00EE` și `2NNN`

Subprogramele funcționează într-un mod similar cu Assembly. Instrucțiunea `2NNN` (*Call*) stochează adresa indicată de program counter (`PC`) în stiva de apel, iar apoi setează `PC`-ul la o altă adresă. Instrucțiunea `00EE` (*Return*) scoate o adresă din stivă și setează `PC`-ul la acea adresă [6].

## 3.5 Tastatura

Calculatoarele pe care `CHIP-8` obișnuia să ruleze utilizau tastaturi hexazecimale, cu 16 taste, de regulă dispuse într-o grilă de 4x4. COSMAC VIP și calculatoarele HP48 utilizau dispunerea tastaturii din Figura 2, pe care o implementează și *Octarou*.

### 3.5.1 `EX9E, EXA1` și `FX0A`

Primele două instrucțiuni sunt de tip *skip* și sar peste instrucțiuni în funcție de tastele apăsate, fără să aștepte apăsarea unei taste.

- `EX9E` sare peste următoarea instrucțiune dacă tasta corespunzătoare valorii din `VX` este apăsată.
- `EXA1` sare peste următoarea instrucțiune dacă tasta corespunzătoare valorii din `VX` NU este apăsată.

Cea de-a treia, `FX0A`, oprește execuția instrucțiunilor (dar nu și decrementarea temporizatorilor) până ce o tastă este apăsată (și eliberată), caz în care valoarea corespunzătoare tastei este pusă în `VX`.


Figura 2: Tastatura de pe COSMAC VIP

## 3.6 Temporizatoarele

Cele două registre `delay` și `sound` funcționează în mod identic. Au mărimea de 1 octet, deci pot reprezenta valori până la 255. Cât timp valorile lor sunt mai mari decât 0, ele sunt decrementate cu 1 de 60 de ori pe secundă, iar acest lucru se întâmplă independent de bucla de execuție a instrucțiunilor. Registrul `sound` cauzează un sunet cât timp valoarea sa este mai mare decât 0.

### 3.6.1 `FX15, FX07` și `FX18`

Aceste instrucțiuni se utilizează pentru a manipula registrele de temporizare. Instrucțiunea `FX15` (*SetDelay*) setează registrul `delay` la valoarea din `VX`, iar `FX18` (*SetSound*) setează registrul `sound` la valoarea din `VX`. `FX07` pune valoarea din `delay` în `VX`.

## 3.7 Instrucțiunile aritmetice și logice

Majoritatea instrucțiunilor aritmetice (Tabela 2) afectează registrul `VF`, excepție făcând `6XNN` și `7XNN`.

### 3.7.1 `6XNN` și `7XNN`

Ambele instrucțiuni manipulează registrele de variabile. *SetLiteral* (`6XNN`) pur și simplu efectuează $\texttt{VX} \leftarrow \texttt{NN}$. *AddLiteral* adună la registrul variabil o valoare ($\texttt{VX} \leftarrow \texttt{VX} + \texttt{NN}$), fără a modifica valoarea din registrul flag `VF`.

### 3.7.2 Registrul `VF`

Instrucțiunea *Add* setează `VF` la 1 daca valoarea maxima (255) este depășită. *Sub* începe prin a seta `VF` la 1, iar apoi, dacă scăzătorul este mai mare decât descăzutul, „împrumută” din `VF`, setându-l la 0.

| Codul | Numele | Operația | Flag register |
|---|---|---|---|
| 8XY0 | *Set* | VX $\leftarrow$ VY | |
| **Operații logice pe biți** | | | |
| 8XY1 | *Or* | VX $\leftarrow$ VX $\vee$ VY | |
| 8XY2 | *And* | VX $\leftarrow$ VX $\wedge$ VY | |
| 8XY3 | *Xor* | VX $\leftarrow$ VX $\oplus$ VY | |
| 8XY6 | *RightShift* | VX $\leftarrow$ VY $\ll 1$ | VF $\leftarrow$ VY $\wedge 1$ |
| 8XYE | *LeftShift* | VX $\leftarrow$ VY $\gg 1$ | VF $\leftarrow$ VY $\gg 7$ |
| **Operații aritmetice** | | | |
| 8XY4 | *Add* | VX $\leftarrow$ VX + VY | VF $\leftarrow$ (VX + VY $> 255$) |
| 8XY5 | *Sub* | VX $\leftarrow$ VX $-$ VY | VF $\leftarrow$ (VY $<$ VX) |
| 8XY7 | *Sub* | VX $\leftarrow$ VY $-$ VX | VF $\leftarrow$ (VX $<$ VY) |

Tabela 2: Operațiile aritmetice CHIP-8

Operațiile *LeftShift* și *RightShift* au comportamente diferite în funcție de varianta CHIP-8. Pe COSMAC VIP, VX lua valoarea lui VY, apoi i se aplica operația de bitshift. Pe CHIP-48 și SUPER-CHIP, VY este ignorat complet, operația aplicându-i-se numai lui VX.

### 3.8 Alte instrucțiuni

Alte două instrucțiuni CHIP-8 sunt CXNN (*Random*) și FX33 (*DecimalConversion*). *Random* generează un număr aleator, apoi pune în VX conjuncția pe biți dintre numărul generat și NN.

*DecimalConversion* ia numărul din VX (un număr de 8 biți, adică $0 \leq$ VX $\leq 255$) și scrie în zona de memorie I..I+3 cele 3 cifre în baza 10 ale acestuia (Algoritmul 2).

**Algoritm 2:** Conversie în baza 10

$n \leftarrow$ VX;
**pentru** $j \leftarrow 3, 0, -1$ **execută**
  $M[I + j] \leftarrow n \mod 10$;
  $n \leftarrow n/10$;
**sfârșit**

## 4 Aplicația

*Octarou* are o structură modulară simplă. Repozitoriul principal conține:

- fișierele de configurare ale mediului de dezvoltare (`flake.nix, Cargo.toml` etc.)
- codul sursă al aplicației, sub `src/`

  modulul `interpreter`, unde se află implementațiile variantelor de CHIP-8

  modulul `app`, unde se află codul pentru interfața grafică

  modulul `main`, punctul de intrare al aplicației
- documentația, incluzând acest document și sursa LaTeX, licența (`EUPL-1.2`) și instrucțiuni de compilare
- programe CHIP-8 (ROMuri), sub `roms/`
- ROMuri pentru testarea interpretorului, sub `tests/`

State-ul aplicației există în structura `Octarou` (Listing 5). Câmpul `interpreter` are tipul `Option<Box<dyn Interpreter>>`, un smart pointer (`Box<T>`) la un trait-object de tip `dyn Interpreter`. Acesta este state-ul interpretorului. Interpretoarele sunt mereu asociate cu un program. Odată cu încărcarea unui program, un nou astfel de obiect este creat, iar cel vechi este eliberat (dacă există). Totodată, pentru a putea susține mai multe variante de `CHIP-8`, în cazul acesta `CHIP-8` și `SUPER-CHIP`, este utlizată o formă de polimorfism bazată pe trăituri.

Traitul `Interpreter` (Listing 6) este implementat de două alte tipuri, anume `Chip8` (Listing 7) și `SuperChip`, fiecare din ele implementând varianta corespunzătoare a limbajului `CHIP-8`. Orice tip care implementează traitul `Interpreter` trebuie să implementeze toate funcțiile asociate acestuia, adică, cu alte cuvinte, orice timp „vrea să fie un interpretor de `CHIP-8`" trebuie să aibă un display, sa își poată actualiza timerele, să citească și să execute o instrucțiune etc.

```
pub struct Octarou {
    interpreter: Option<Box<dyn Interpreter>>,
    mode: Mode,
    speed: u64,
    current_program: Option<Program>,

    screen_size: egui::Vec2,
    current_tab: Tab,

    file_dialog_channel: (mpsc::Sender<Program>, mpsc::Receiver<Program>),

    #[allow(unused)]
    stream: (rodio::OutputStream, rodio::OutputStreamHandle),
    sink: rodio::Sink,
    muted: bool,
}
```

Listing 5: State-ul aplicației

## 4.1 Interpretarea `CHIP-8`

Un interpretor de `CHIP-8`, în principiu, execută trei pași într-o buclă infinită: citește o instrucțiune din memorie, decodează instrucțiunea pentru a afla ce trebuie să facă, iar apoi execută instrucțiunea [6]. Exterior acestei bucle, temporizatoarele trebuie actualizate cu rata de 60 Hz. Bucla de execuție trebuie executată și ea cu o anumită viteză, întrucât, dacă ar fi executată prea repede, programele și jocurile `CHIP-8` ar fi greu sau imposibil de folosit. Prin urmare, ea rulează implicit de 700 de ori pe secundă, însă aceast parametru poate fi modificat prin interfață grafică a aplicației.

### 4.1.1 Citirea și decodarea instrucțiunilor

Citirea și decodarea instrucțiunilor se realizează cu ajutorul funcției `next_instruction` (Listing 8). Ea este responsabilă pentru a extrage și interpreta instrucțiunile din memoria interpretorului `CHIP-8`. Aceasta începe prin extragerea următorului opcode (cod operațional) din memoria sistemului, interpretând 2 octeți (16 biți) de la adresa curentă a programului. Apoi, avansând adresa programului cu 2 pentru a indica următoarea instrucțiune, funcția încearcă să creeze o nouă instrucțiune Chip8 folosind opcode-ul extras. În cazul în

```rust
pub trait Interpreter {
    fn display(&self) -> Vec<&[u8]>;
    fn is_beeping(&self) -> bool;

    fn update_timers(&mut self);
    fn next_instruction(&mut self) -> Result<Instruction, InterpreterError>;
    fn execute_instruction(
        &mut self,
        instruction: Instruction,
        keys_pressed: &[bool; 16],
        keys_released: &[bool; 16],
    ) -> Result<(), InterpreterError>;

    fn tick(
        &mut self,
        keys_down: &[bool; 16],
        keys_released: &[bool; 16],
        speed: u64,
    ) -> Result<(), InterpreterError> {
        // auto-implementation
    }
}
```

Listing 6: Traitul `Interpreter`

```rust
pub struct Chip8 {
    memory: [u8; MEMORY_SIZE],
    pc: usize,
    index: usize,
    stack: Vec<usize>,
    delay_timer: u8,
    sound_timer: u8,
    variables: [u8; 16],
    display: [[u8; DISPLAY_WIDTH]; DISPLAY_HEIGHT],
}
```

Listing 7: Structura interpretorului de `CHIP-8`

care opcode-ul este necunoscut sau adresa indicată depășește memoria disponibilă, funcția returnează varianta `Err` a tipului `Result<Instruction, InterpreterError>`.

Tipul `Instruction` este o enumerație (un tip de date algebric) care memorează diferitele tipuri de instrucțiune. Semnătura funcției care parsează codul operațional este

```rust
pub fn new(opcode: u16) -> Option<Self>
```

În cazul în care primește un cod invalid, aceasta va întoarce varianta `None`, care va semnifica apelatorului că nu există o instrucțiune asociată acelui cod.

### 4.1.2 Executarea instrucțiunilor

Funcția `execute_instruction` este responsabilă pentru executarea instrucțiunilor interpretate din memoria interpretorului `CHIP-8`. Aceasta primește o instrucțiune și starea curentă a tastelor, apoi, în funcție de tipul instrucțiunii, efectuează operațiile corespunzătoare.

```rust
fn next_instruction(&mut self) -> Result<Instruction, InterpreterError> {
    let opcode = u16::from_be_bytes(
        self.memory
            .get(self.pc..self.pc + 2)
            .ok_or(InterpreterError::OutOfMemory)?
            .try_into()
            .expect("Slice should always have length 2"),
    );

    self.pc += 2;
    Instruction::new(opcode).ok_or(InterpreterError::UnknownOpcode {
        opcode,
        address: self.pc,
    })
}
```

Listing 8: Funcția pentru citirea și decodarea instrucțiunilor

## 4.2 Extensia SUPER-CHIP

În anul 1990, Erik Bryntse a scris un emulator Chip8 pentru calculatorul grafic HP-48 numit CHIP-48, care adaugă o serie de instrucțiuni extinse denumite SCHIP sau SUPER-CHIP. SUPER-CHIP își propune să fie compatibil cu versiunea obișnuită a setului de instrucțiuni Chip8, iar noile instrucțiuni ocupă spațiul neutilizat din codificarea instrucțiunilor Chip8 [1].

*Octarou* implementează SUPER-CHIP și folosește polimorfismul pentru a permite încărcarea dinamică a programelor într-un interpretor de SUPER-CHIP.

În ceea ce privește noutățile aduse de SUPER-CHIP față de CHIP-8, există mai multe îmbunătățiri și extinderi ale setului de instrucțiuni. În primul rând, SUPER-CHIP adaugă un ecran mai mare, de dimensiuni 128x64 pixeli, față de ecranul de 64x32 pixeli al CHIP-8. De asemenea, SUPER-CHIP include un set suplimentar de instrucțiuni grafice, cum ar fi desenarea sprite-urilor mai mari, de dimensiuni de 16x16 pixeli. Iată instrucțiunile noi adăugate în SUPER-CHIP [4].

- `00FF` (`Hires`) – Activează modul grafic de înaltă rezoluție 128x64.
- `00FE` (`Lores`) – Dezactivează modul grafic de înaltă rezoluție și revine la 64x32.
- `00CN` (`ScrollDown {amount: usize}`) – Derulează ecranul în jos cu 0 până la 15 pixeli.
- `00FB` (`ScrollRight`) – Derulează ecranul la dreapta cu 4 pixeli.
- `00FC` (`ScrollLeft`) – Derulează ecranul la stânga cu 4 pixeli.

## 4.3 Interfața grafică

Interfață grafică este implementată prin librăriile `eframe` și `egui`. Este împărțită în două panouri: unul în stânga, unde se află meniul și alte controale ale interpretorului, și unul principal, unde se află afișajul interpretorului. Ecranul principal găzduiește și o listă cu mesaje de logging, accesabilă prin intermediul meniului cu taburi din partea de sus a ferestrei. Butonul *Menu* permite deschiederea unui dialog nativ al sistemului de operare prin care este posibilă încărcarea programelor în interpretor.

```
ScrollRight => {
    let amount = match self.hires {
        true => 4,
        false => 2 * 4,
    };

    self.display.iter_mut().for_each(|row| {
        row.rotate_right(amount);
        row[0..amount].fill(0);
    });
}
ScrollLeft => {
    let amount = match self.hires {
        true => 4,
        false => 2 * 4,
    };

    self.display.iter_mut().for_each(|row| {
        row.rotate_left(amount);
        row[DISPLAY_WIDTH - amount..].fill(0);
    });
}
ScrollDown { amount } => {
    let amount = match self.hires {
        true => amount,
        false => 2 * amount,
    };
    self.display.rotate_right(amount);
    self.display[0..amount].fill([0; DISPLAY_WIDTH]);
}
```

Listing 9: Implementația derulării pentru `SUPER-CHIP` în modul `hires`


Figura 3: *Octarou* rulând un program simplu


Figura 4: *Octarou* rulând testele `corax+`

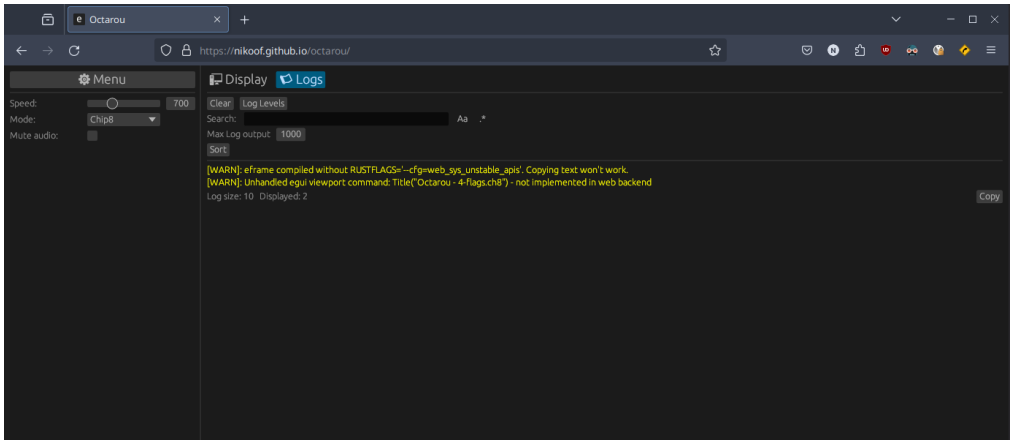


Figura 5: Lista cu mesaje de logging a *Octarou*, rulând în Mozilla Firefox

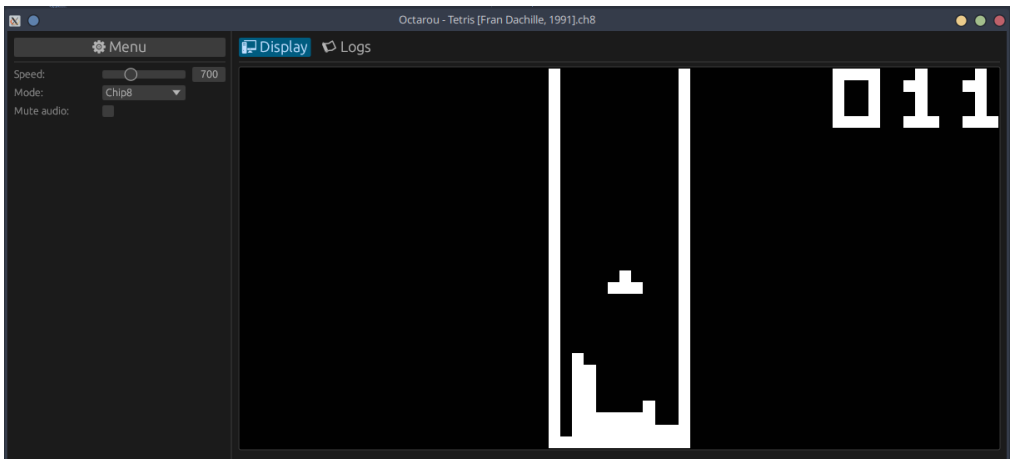


Figura 6: *Octarou*, rulând în faimosul joc *Tetris*

# 5 Îmbunătățiri

Una din primele îmbunătățiri pe care aș vrea să le implementez este suportul pentru `XO-CHIP`, o variantă modernă de `CHIP-8` folosită în *Octo*, un compilator și IDE pentru `CHIP-8`. Majoritatea programelor noi sunt scrise pentru `XO-CHIP`, ceea ce ar însemna ca interpretorul meu ar putea rula mult mai multe programe interesante.

Un alt lucru ce trebuie îmbunătățit este suportul pentru web, întrucât, versiunea 1.1.0 nu implementează temporizarea exactă, întrucât API-urile pentru timp nu sunt implementate complet în ecosistemul WebAssembly. De aceea, deși este funcțional, experiența de utilizator pe versiunea web a *Octarou* nu este întocmai plăcută.

În afară de acestea, mereu există opțiunea să adaug mai multe chestiuni configurabile legate de diferitele variante de `CHIP-8`. Util ar fi și un sistem de depanare a programelor. Posibilitățile în acest sens sunt numeroase.

# 6 Concluzii

În procesul de creare a acestui proiect, am învățat multe lucruri despre programarea în limbajul Rust și despre arhitectura sistemelor CHIP-8 și SUPER-CHIP. Crearea unui interpretor pentru aceste sisteme a necesitat o înțelegere profundă a funcționării lor interne, precum și abilități solide de analiză și implementare a instrucțiunilor specifice. Am fost expus la concepte precum manipularea memoriei, interpretarea instrucțiunilor și gestionarea stării sistemului.

Unul dintre cele mai valoroase aspecte ale procesului de dezvoltare a acestui interpretor a fost navigarea prin provocările de optimizare și gestionare a performanței, în special în ceea ce privește ciclul de execuție al instrucțiunilor. De asemenea, am experimentat cu diverse abordări de structurare a codului și de gestionare a dependențelor într-un mod eficient și modular. Folosind Nix, am putut crea un mediu de dezvoltare izolat și reproductibil, care a facilitat gestionarea dependențelor și a permis implementarea fără prea mult efort a unui pipeline de CI (*continuous integration*).

De asemenea, prin lucrul la acest proiect am avut ocazia să contribui la un proiect sursă deschisă [7].

Acest proiect nu numai că mi-a permis să explorez concepte avansate de programare și arhitectură de calculatoare, dar mi-a oferit și oportunitatea de a îmi îmbunătăți abilitățile de dezvoltare software. Dezvoltarea de interpretoare și emulatoare reprezintă nu doar un exercițiu tehnic, ci și o explorare a istoriei calculatoarelor și a fundațiilor programării, punând în lumină importanța continuă a învățării și a evoluției în acest domeniu în continuă schimbare.

# 7 Bibliografie


[1] Erik Bryntse, *SUPER-CHIP v1.1 (now with scrolling)*, 28 Mai 1991, URL: `http://devernay.free.fr/hacks/chip8/schip.txt` (accesat în 12.2.2024).

[2] Eelco Dolstra, „The Purely Functional Software Deployment Model", Teză de doct., Universitatea din Utrecht, Ian. 2006.

[3] Emil Ernerfeldt, *egui: an easy-to-use GUI in pure Rust*, URL: `https://github.com/emilk/egui` (accesat în 13.2.2024).

[4] CHIP-8 Research Facility, *CHIP-8 extensions and compatibility*, URL: `https://chip-8.github.io/extensions/` (accesat în 12.2.2024).

[5] Steve Klabnik, Carol Nichols și contribuitorii din Comunitatea Rust, *The Rust Programming Language*, No Starch Press, URL: `https://doc.rust-lang.org/stable/book/`.

[6] Tobias V. Langhoff, *Guide to making a CHIP-8 emulator*, URL: `https://tobiasvl.github.io/blog/write-a-chip-8-emulator/` (accesat în 12.2.2024).

[7] Jacob Regen și Nicolas Bratoveanu, *egui_logger v0.4.3*, URL: `https://github.com/RegenJacob/egui_logger/releases/tag/v0.4.3` (accesat în 15.2.2024).

[8] *Rust Programming Language*, URL: `https://www.rust-lang.org/` (accesat în 13.2.2024).

[9] Steffen Schümann, *CHIP-8 Variant Opcode Table*, URL: `https://chip8.gulrak.net/` (accesat în 12.2.2024).


# 8 Index

## Listinguri de cod sursă



## Listă de figuri