book

Babeş–Bolyai Tudományegyetem Kolozsvár
Matematika és Informatika Kar
Informatika Szak

Szakdolgozat

Szakdolgozat cím

Témavezető:
dr. Bodó Zalán

Szerző:
Zediu
Álmos-Ágoston

2022

Babeş–Bolyai University of Cluj-Napoca
Faculty of Mathematics and Informatics
Specialization: Computer Science

Diploma Thesis

License thesis title

Advisor:
dr. Bodó Zalán

Author:
Álmos-Ágoston Zediu

2022

Universitatea Babeş–Bolyai, Cluj-Napoca
Facultatea de Matematică şi Informatică
Specializarea Informatică

Lucrare de licenta

Titlu lucrare licență

Conducător ştiinţific:
dr. Bodó Zalán

Absolvent:
Álmos-Ágoston
Zediu

2022

1.

Bevezetés

2.

Clojure

A Clojure programozási nyelv egy dinamikus funkcionális nyelv, mely ötvözi a JVM platform előnyeit a Lisp nyelvek kifejezőkészségével.

2.1 Funkcionális programozás Clojureben

A Clojureben a függvények az elsőrendű absztrakciók, képesek vagyunk akár argumentumként is kezelni őket, stb.

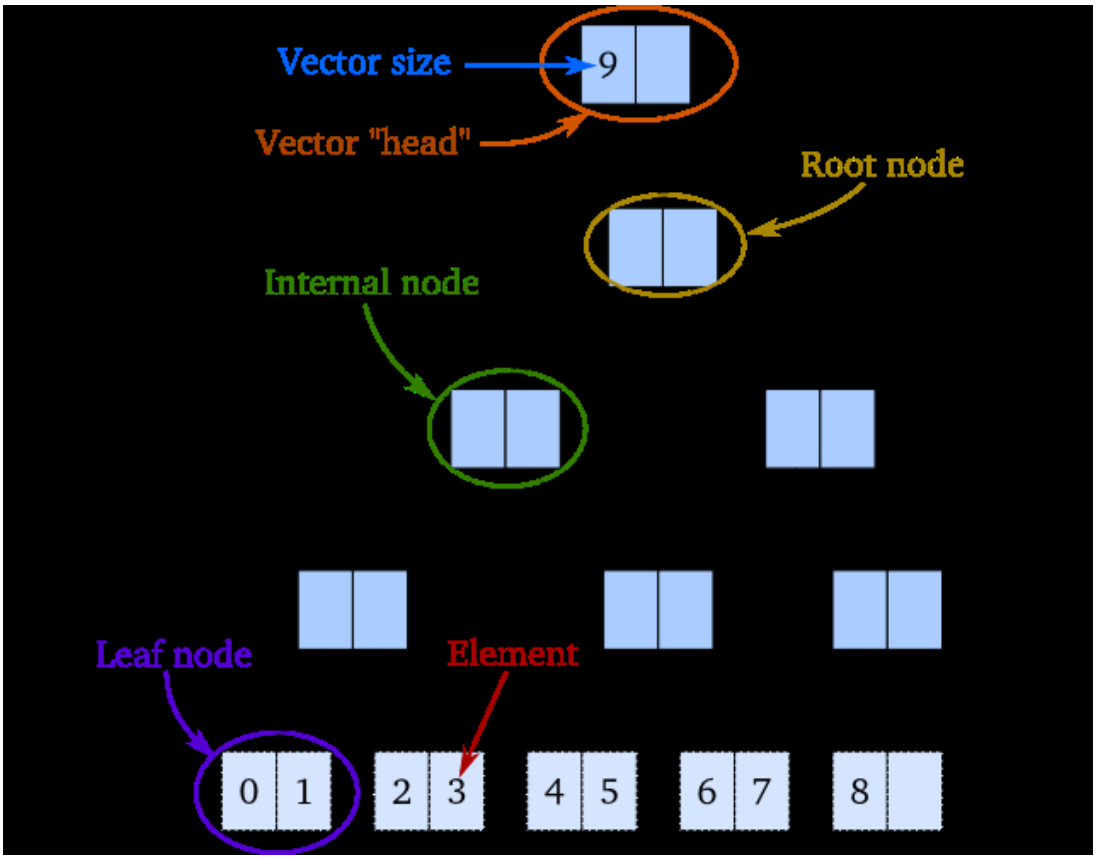
```
(defn my-adder [a b]
  (+ a b))

(def my-five-adder (partial my-adder 5))
(map my-five-adder [1, 2, 3, 4])
```

- (“#’user/my-adder”)
- (“#’user/my-five-adder”)
- (“(6 7 8 9)”)

2.2 Perzisztens adatstruktúrák

Rich Hickey az adatstruktúráit az ideális hasítófákra alapozta (Bagwell, 2001). Egy konceptuális elképzelésért rátekinthetünk erre a képre:



A lényegi rész az, hogy ahhoz, hogy olyan adatstruktúrák, mint a vektorok performánsak legyenek, de perzisztensek, szükségünk van specializált bináris fák felépítésére.

2.3 Homoikonicitás

Ami talán leginkább megkülönbözteti a Lisp nyelvcsaládban levő nyelveket a többiektől, az a homoikonicitás (McIlroy, 1960) tulajdonság, vagyis maga a programkód formálható ugyanazzal a nyelvvel futás közben, mint amiben meg volt írva.

Hasonló viselkedést elérhetünk nem homoikonikus nyelvekben is, mint mondjuk a Java vagy a C# reflection rendszere, vagy pedig a Python dekorátor szintaxisa, viszont a Lisp nyelvek makrórendszereivel azért könnyebb valamilyen szinten dolgozni, mivel nincsenek speciálisan megkülönböztetve a programban felhasznált adatstruktúrák szintaxisai, és a programot felépítő, elágazásokat, ismétlő ciklusokat, függvénydefiníciókat jelző nyelvi struktúrák szintaxisai.

Vegyük példának okáért a következő egyszerű programot:

```
(defn add-list-numbers [number-list]
  (apply + number-list))

(add-list-numbers '(1 2 3 4 5))
```

- ("#'user/add-list-numbers")
- ("15")

Látható, hogy a függvénydefiníció kerek zárójelekbe írtuk, a függvény argumentumai pedig egy vektorszerű struktúrában kaptak helyet, utána pedig maga a függvényhívás is zárójelek között volt. Érdekes módon az átadott lista szintúgy zárójelezve adódott át, viszont raktunk elé egy aposztrófot is.

Erre azért volt szükség, mivel a Lisp nyelvekben a kerek zárójel listát jelöl, és minden lista, hacsak nem jelezzük aposztrófal, függvénymeghívással jár. Annak köszönhetően viszont, hogy "listákban" programozunk, képesek vagyunk a programrészleteinket mint lista, vektor, vagy halmazelemeket módosítani átrendezni.

2.3.1 Makrók

A Lisp makrók olyan programszerkezetek, amelyek egy programrészletet kapnak meg, módosítják azt, és a módosított programrészlet eredményét futtatják végül le. Fontos megjegyezni, hogy ez fordítási időben történik, nem futási időben.

1. **TODO** Ezt még átfogalmazni picit Egy jó példa arra, hogyan segíthet ez fejlesztésben és talán még fontosabb, adatelemzés során, az az úgynevezett "threading" makró.

   ```
   (defn generate-masked-grouped-ratings [dataset-path]
     (-> (load-ratings dataset-path)
         (tc/dataset)
         (tc/complete :user :item)
         (tc/group-by :user {:result-type :as-seq})))
   ```

   Ez a makró be van építve a nyelvbe, és forráskódja is rövid.

   ```
   (defmacro ->
     [x & forms]
     (loop [x x, forms forms]
       (if forms
   ```

```
(let [form (first forms)
      threaded (if (seq? form)
                 (with-meta `(~(first form) ~x ~@(next form)) (meta form))
                 (list form x))]
  (recur threaded (next forms)))
x)))
```

3.

Algoritmusok

3.1 Locality sensitive hashing

Lehet beszélni erről a (Charikar), vagy pedig,

3.2 SVD

(Brand, 2003)

Bibliography

Bagwell, P., editor. *Ideal Hash Trees*. 2001.

Brand, M. Fast online SVD revisions for lightweight recommender systems. In *Proceedings of the 2003 SIAM International Conference on Data Mining*, pages 37–46. Society for Industrial and Applied Mathematics, May 2003. ISBN 978-0-89871-545-3 978-1-61197-273-3. doi: 10.1137/1.9781611972733.4.

Charikar, M. S. Similarity Estimation Techniques from Rounding Algorithms. page 9.

McIlroy, M. D. Macro instruction extensions of compiler languages. *Communications of the ACM*, 3(4):214–220, Apr. 1960. ISSN 0001-0782. doi: 10.1145/367177.367223.