

Proiect de atestat - Interpretor de Pseudocod

Realizator: Cîcu Timotei-Iosif

2023

Detalii de implementare a interpretorului de pseudocod, detalii despre utilizarea interpretorului, si exemple de pseudocod valide.

1. Introducere

Am ales sa realizez acest proiect, deoarece sunt interesat in crearea limbajelor de programare compilate si interpretate.

Interpretorul a fost creat cu scopul de a invata despre limbajele de programare interpretate.

2. Prezentarea implementarii interpretorului de pseudocod

Am realizat interpretorul folosind in limbajul de programare C. Acesta are patru etape:

2.1. Etapa de lexare/tokenizare

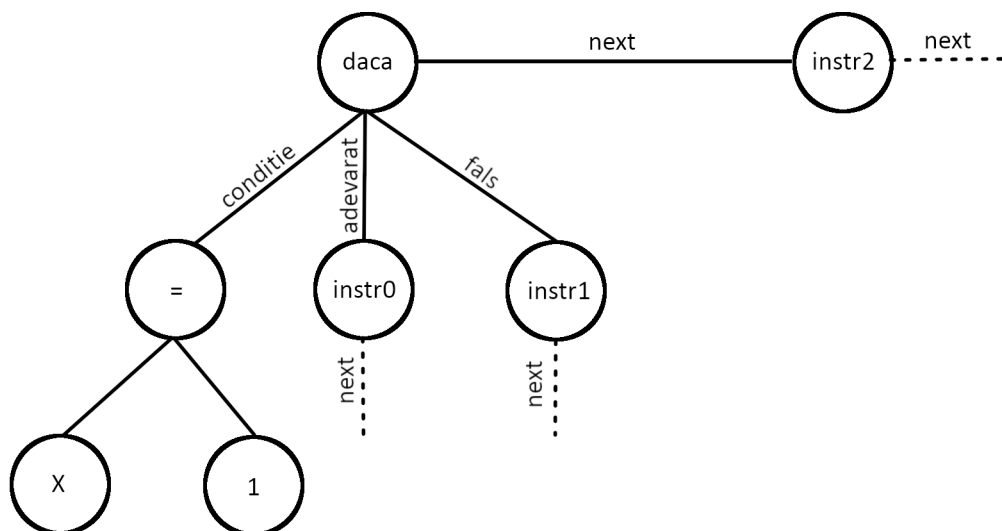
Interpretorul citeste fisierul de intrare, apoi parcurge caracter cu caracter codul citit, transformandu-l intr-o serie de tokenuri care vor fi utilizate in etapa urmatoare (etapa de parsare).

2.2. Etapa de parsare

Programul parcurge tokenurile obtinute din etapa anterioara si formeaza un arbore de executare care este utilizat la etapa urmatoare.

Expresiile aritmetice sunt transformate din scrierea infix, in scriere prefix.

Exemplu de parte de arbore de executare:



2.3. Etapa de executare

Programul parcurge arborele de executare pentru a executa instructiunile scrise in fisierul de intrare.

Apeluri de functii

Atunci cand o functie este apelata, se copiaza intr-un hash map de variabile paramentrii functiei, iar acest hash map este pus (push) intr-un stack (stiva).

Atunci cand o functie returneaza sau se termina, se scoate (pop) din stack (stiva), memoria alocata pentru hash mapul nou de variabile este eliberata si memoria alocata pentru noul element in stack (stiva) este si ea eliberata.

Variabile

Variabilele locale sunt memorate in hash mapul de variabile asociat capului stivei, iar variabilele globale sunt memorate in hash mapul `global_var_map`.

Atunci cand nu s-a apelat nicio functie, hash mapul de variabile locale din stack (stiva) este acelasi cu hash mapul de variabile globale `global_var_map`.

2.4. Etapa de garbage collection:

Variabilele de tip vector si de tip functie sunt alocate dinamic, iar atunci cand o variabila isi schimba valoarea si nu mai are referinta catre inceputul vectorului/catre functie, se poate crea un "memory leak".

Atunci cand un vector sau o functie sunt create, adresa lor de memorie este inserata intr-un hash map pentru a putea tine minte locatia lor.

Atunci cand o variabila primeste prin assignment, citire, sau apelare la functie o adresa de memorie care a fost alocata dinamic, hash mapul este actualizat si frecventa de cate variabile au referinta catre acea adresa creste.

Atunci cand o variabila care retinea o asemena adresa primeste prin assignment sau citire o noua valoare, sau se returneaza din functie si se elibereaza hash mapul de variabile locale, numarul de referinta catre acea parte de memorie scade.

Pointerile returnate de la functia `referinta(x)`, nu sunt contorizate.

Hash mapul utilizat pentru garbage collection este un hash map de frecventa

Atunci cand numarul de referinte a scazut pana la 0, se verifica cate alte adrese au un numar de referinte egal cu 0 si aceste adrese se elibereaza.

Exemple de cod in care s-ar elibera memorie folosind garbage collection

```
x <- vector(3)
x <- 10
```

sau

```
x <- vector(3)
citeste x
```

sau

```
subprogramul f() face
|   x <- vector(3)
f()
```

Implementarea hash mapului

Si `variable_map` si `possible_garbage_map` folosesc aceeasi structura de date generalizata, folosind functii pentru utilitatiile speciale (o functie de hash, de indexare, de dereferentiere o singura data, de inserare in lista inlantuita, si de cautare in lista inlantuita)

3. Compilare si utilizare a interpretorului

3.1. Compilare

Interpretorul se compileaza utilizand compilatorul `gcc` (sau echivalent) folosind urmatoarea comanda: `gcc main.c -lm -o pseudocod`

3.2. Utilizare

Interpretorul se apeleaza folosind comanda `pseudocod [fisier]` - unde `fisier` este fisierul care contine codul sursa

4. Sintaxa utilizata

4.1. Indentare

Indentarea sa face folosind caracterul `|`.

Exemplu:

```
daca 1 atunci
|   scrie "da"
```

Daca la sfarsitul liniei anterioare s-a folosit caracterul de escape `\`, atunci nu se va mai indenta folosind, deoarece interpretorul trateaza acest caz ca si cand nu ar fi o linie noua intre cele doua linii

Exemplu:

```
daca 1 si \  
    2 atunci  
|   scrie "da"
```

Instructiunile de decizie, repetitive, si definirea functiilor au nevoie de indentare pentru a determina codul din interiorul structurii.

4.2. Expresiile (instructiunile aritmetice)

Expresiile sunt combinari valide de operatori unari si binari, variabile, valori, si apeluri la functii

Operaturul unar este not, iar operatorii binari sunt: <- (assignment), sau, si, =, !=, <, >, <=, >=, +, -, *, /, % (restul impartirii), ^ (ridicarea la putere).

Ordinea operatorilor

- Ordinul 9: not
- Ordinul 8: ^
- Ordinul 7: *, /, % (restul impartirii)
- Ordinul 5: +, -
- Ordinul 4: <, >, <=, >=
- Ordinul 3: =, !=
- Ordinul 2: si
- Ordinul 1: sau
- Ordinul 0: <- (assignment)

Operatiile de ordin mai mare vor fi executate primele.

Exemple de expresii valide:

```
x <- 10 * 2 / 2 * f(x)
```

```
x = x != x
```

```
x <- -1
```

Exemple de expresii invalide

```
x <- 10 * 2 / 2 *
```

```
x = != x
```

```
x <- * 1
```

4.3. Variabilele

Variabilele sunt declarate prin asignare, citire, sau prin definirea unor functii (functiile definite de utilizator sunt variabile).

Numele variabilelor pot sa contina litere, caracterul. _ (underscore), si numere, dar nu pot sa inceapa cu numere.

4.4. Tipurile de variabile

Tipurile de variabile sunt: TV_NUMAR (numere de tip intreg si real), TV_CHAR (caractere), TV_VECTOR (vectori de alte tipuri de variabile), TV_TIP (returnat de functia tip()), TV_ERROR (returnat de functia tip()).

Prescurtarea TV vine de la "tip variabila".

Functia tip() este explicata la sectiunea despre functiile predefinite.

4.5. Declararea si apelarea functiilor

Functiile sunt definite utilizand cuvantul cheie subprogramul, urmat de numele subprogramul, lista de

parametrii in paranteze, si cuvantul cheie face astfel:

```
subprogramul nume_subprogram(p0, p1, p2, ...) face
|
|   cod
```

Funcțiile sunt apelate folosind numele subprogramului si lista de variabile sau valori in paranteze astfel:

```
nume_subprogram(v0, v1, v2, ...)
```

Daca functia nu are parametrii, atunci se va declara astfel:

```
subprogramul nume_subprogram() face
|
|   cod
```

Si se va apela astfel:

```
nume_subprogram()
```

4.5.1. Functii predefinite

- `sqrt(x)` - radical din x
- `floor(x)` - da partea intreaga a lui x
- `tip(x)` - returneaza tipul lui x, valoarea returnata are tip `TV_TIP`
- `vector(x)` - alocheaza memorie pentru a crea un vector de dimensiune x
- `dimensiune(x)` - da dimensiunea vectorului x
- `referinta(x)` - returneaza o referinta catre x sub forma unui vector cu dimensiune 1, functie folosita pentru a modifica valoarea unei variabile neglobale in interiorul unui subprogram. se derefentiaza prin accesarea indexului 0
- `cast(x, tip)` si `schimbare_tip(x, tip)` - returneaza valoarea lui x, dar cu tip `tip`, `tip` este o variabila sau o constanta de tip `TV_TIP`

4.6. Instructiunea de decizie

Instructiunea de decizie se foloseste astfel:

```
daca expr0 atunci
|
|   cod daca expr0 este adevarat
```

Instructiunea de decizie cu doua ramuri:

```
daca expr0 atunci
|   cod daca expr0 este adevarat
|altfel
|   cod daca expr0 este fals
```

Instructiunea de decizie cu mai mult de doua ramuri:

```
daca expr0 atunci
|       cod daca expr0 este adevarat
|altfel daca expr1 atunci
|       |       cod daca expr0 este fals si expr1 este adevarat
|       |altfel
|       |       cod daca expr0 este fals si expr1 este fals
```

Observatie: Spre deosebire de celelalte structuri dupa care trebuie sa urmeze indentatie, dupa cuvantul cheie `altfel` poate urma o expresie/instructiune direct.

4.7. Instructiuni repetitive cu numar nedeterminat de pasi

4.7.1. Instructiuni repetitive cu test initial

Instructiunile repetitive cu test initial vor executa codul din interiorul structurii doar daca expresia este adevarata.

4.7.1.1. Instructiunea cat timp ... executa

Instructiunea cat timp ... executa se foloseste astfel:

```
cat timp expr0 executa
|   cod daca expr0 este adevarat
```

4.7.1.2. Instructiunea pana cand ... executa

Instructiunea pana cand ... executa se foloseste astfel:

```
pana cand expr0 executa
|   cod daca expr0 este fals
```

4.7.2. Instructiuni repetitive cu test final

Instructiunile repetitive cu test final vor executa cel putin o data codul din structura, indiferent daca expresia este adevarata, apoi vor executa codul din nou, doar daca expresia este adevarata

4.7.2.1. Instructiunea repeta ... cat timp

Instructiunea repeta ... cat timp se foloseste astfel:

```
repeta
|   cod daca expr0 este adevarat
| cat timp expr0
```

4.7.2.2. Instructiunea repeta ... pana cand

Instructiunea repeta ... pana cand se foloseste astfel:

```
repeta
|   cod daca expr0 este fals
| pana cand expr0
```

4.8. Instructiunea repetitiva cu numar determinat de pasi

4.8.1. Instructiunea pentru ... executa

Instructiunea pentru ... executa se foloseste astfel:

```
pentru expr0, expr1, expr2 executa
|   cod daca expr1 este adevarata
```

expr0 va fi executata, apoi se va testa daca expr1 este adevarata si daca este adevarata, se va executa codul din structura, la sfarsitul codului din structura, se va executa expr2 si se va intoarce la testare.

Observatie: spre deosebire de pseudocodul invatat la scoala, aceasta instructiune are aceeasi functionalitate ca si un for din C.

4.9. Instructiuni de intrare/iesire

4.9.1. Instructiuni de intrare

4.9.1.1. Instructiunea citestetot

Instructiunea citestetot va fi urmata de o variabila si se va citi de la tastatura un sir de caractere pana la caracterul de linie noua si se va stoca in variabila sub forma unui vector de caractere.

Exemplu:

```
citestetot x
```

4.9.1.2. Instructiunea citeste

Instructiunea citeste va fi urmata de un numar nedeterminat de variabile si se va citi de la tastatura cuvant cu cuvant si se va determina daca poate fi interpretat ca numar. Daca poate fi interpretat ca numar, atunci va fi memorat ca numar (TV_NUMAR) in variabila corespunzatoare numarului cuvantului. Daca nu poate fi interpretat ca numar, atunci se va stoca in variabila corespunzatoare ca un vector de caractere.

Exemplu:

```
citeste x, y, z
```

Si se citeste de la tastatura:

```
abc 123 b1
```

Atunci x va memora vectorul de caractere “abc”, y va memora numarul 123, z va memora vectorul de caractere “b1”.

Exemplu:

```
citeste x
```

Si se citeste de la tastatura:

```
12.12
```

Atunci x va memora numarul 12.12 (doisprezece virgula doisprezece).

4.9.2. Instructiuni de iesire

4.9.2.1. Instructiunea scrie

Instructiunea `scrie` va fi urmata de un numar nedeterminat de variabile si valori care vor fi afisate in consola in functie de tipul lor:

- `TV_NUMAR` - va afisa numarul
- `TV_CHAR` - va afisa caracterul
- `TV_VECTOR` - daca este un sir de caractere, atunci se va afisa sirul de caractere. Daca nu este sir de caractere, atunci interpretorul va da eroare
- `TV_TIP` - va afisa `TV_NUMAR`, `TV_CHAR`, `TV_VECTOR`, `TV_TIP`, sau `TV_ERROR` in functie de valoarea variabilei
- `TV_ERROR` - se va trata precum variabila are tip de numar (`TV_NUMAR`)

4.10. Vectori

4.10.1. Declarea vectoriilor

Vectorii pot fi declarati in doua moduri.

```
x <- vector(3)
```

care va creea un vector de dimensiune 3

si

```
x <- (1, 2, 3)
```

care va creea un vector de dimensiune 3 cu valorile 1, 2, 3.

Vectorii pot avea mai multe dimensiuni astfel:

```
x <- vector(3)
x[0] <- vector(3)
```

si

```
x <- ((1, 2, 3), 2, 3)
```

4.10.2. Indexarea vectoriilor

Vectorii pot fi indexati utilizand urmatoarea sintaxa:

```
x[expr0][expr1]...[exprn]
```

Exemplu:

```
x <- ((1, 2, 3), 2, 3)
scrie x[0][1], " ", x[2]
```

va afisa:

```
2 3
```

4.11. Cuvatul cheie sursa

Acest cuvânt cheie este singurul care este interpretat în etapa de lexing, similar cu C. Codul din noul fișier este inserat în lista de tokenuri exact în locul în care apare, iar mai târziu este tratat la fel ca și codul din sursa principală.

Dupa cuvântul cheie sursa, se va scrie locația fișierului care să fie inclus.

Exemplu:

Fișierul main.pseudo

```
sursa "alta_sursa.pseudo"  
f()
```

Fișierul alta_sursa.pseudo

```
subprogramul f() face  
|   scrie "123"
```

Va afișa:

```
123
```

4.12. Comentarii

Comentariile pot fi scrise utilizând caracterul # similar cu python. Tot ce urmează după caracterul # va fi ignorat până la următoarea linie.

Exemplu:

```
# daca 1 atunci  
# | scrie 2
```

Nu va afișa nimic.

5. Exemple de cod în directiva "examples"

Am realizat câteva programe folosind pseudocodul/limbajul de programare inventat de mine pentru a afișa abilitățile acestuia.

5.1. X și 0

Am implementat jocul de x și 0 în pseudocod folosind citirea și afișarea în consolă. Tabla de joc este memorată într-un vector/tablou bidimensional.

Atunci când trebuie intrată în casuta în care se va pune simbolul (x sau 0), se va afișa un prompt care evidențiază a cui tură este. Inputul este reprezentat de un număr între 1 și 3 pentru linie și un număr între 1 și 3 pentru coloană, acestea fiind separate printr-un spațiu.

Programul va determina automat cazurile de câștig sau egalitate.

5.2. Sah

Am implementat o versiune incompletă a jocului de sah în pseudocod folosind citirea și afișarea în consolă. Tabla de sah este memorată într-un vector/tablou bidimensional.

Acesta are aproape toate mișcările, cu excepția mișcării "en passant". Acesta poate detecta egalitate doar în cazurile stalemate, și 50 move rule, dar nu și în cazurile dead position rule, sau threefold repetition. În rest, jocul este complet.

Programul va detecta automat saș-ul și sașmat-ul.

Mișcările se scriu la promptul mutare>. Mișcările nu se specifică folosind notația de saș, ci se specifică casuta de început a piesei și unde trebuie mutată folosind sintaxa literă-număr (coloană-linie; exemplu: D2 D4). Programul va detecta automat dacă mutarea este validă.

În cazul în care o mișcare este invalidă, programul va explica criteriul folosit pentru a detecta că mișcarea este invalidă. Tabla de saș este afișată la fiecare tură din perspectiva persoanei care trebuie să mute.

5.3. Probleme de eficienta

Am rezolvat o parte (10 probleme) din problemele propuse la proba practica pentru examenul de atestare a competentelor profesionale a absolventilor claselor de matematica-informatica si matematica-informatica, intensiv informatica, in pseudocod.

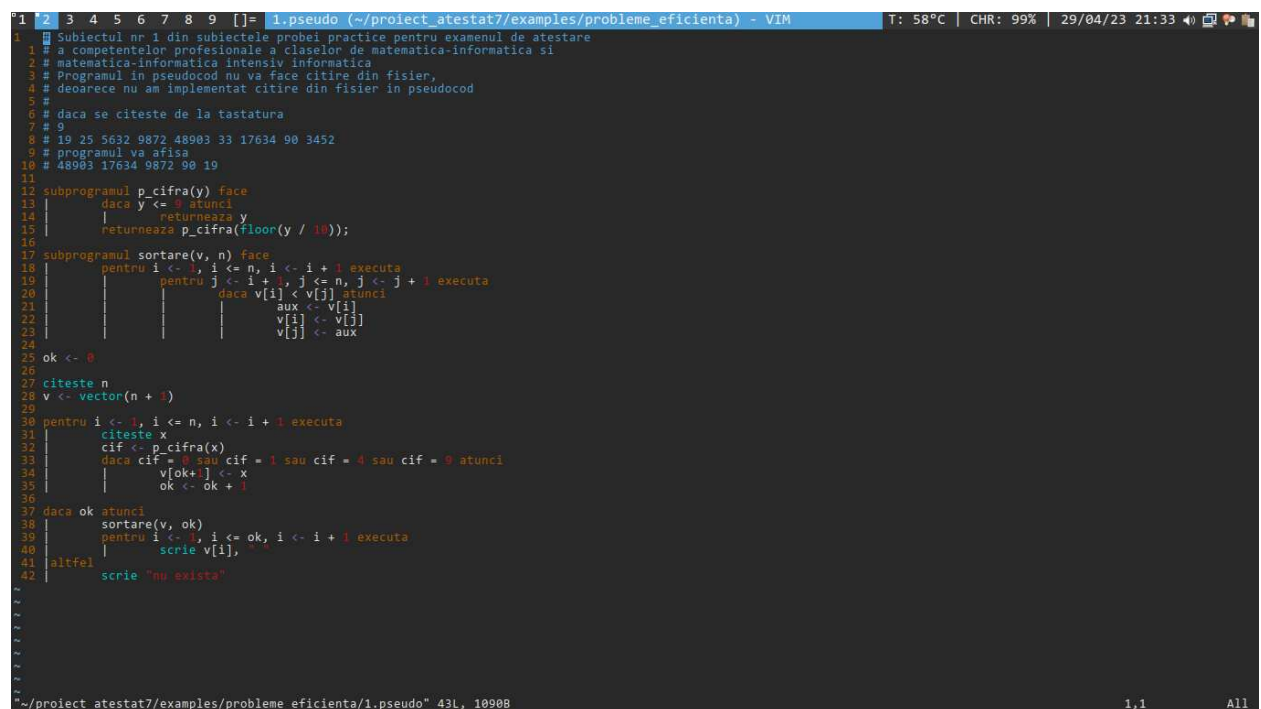
Deoarece nu am implementat citire si afisare in fisier in pseudocod, am modificat cerinta unor probleme, astfel incat sa foloseasca citirea de la tastatura si afisarea in consola (stdin si stdout).

6. Pentru utilizatorii de linux

Programul se poate instala folosind comanda `sudo make install`

7. Pentru utilizatorii de vim

In folderul/directiva `vim` am inclus fisierul `pseudo.vim` pentru sintaxa astfel incat vim sa foloseasca highlightarea de sintaxa pentru pseudocod, de exemplu:



```

1 2 3 4 5 6 7 8 9 []= 1.pseudo (~/.project_atestat7/examples/probleme_eficienta) - VIM
1 # Subiectul nr 1 din subiectele probei practice pentru examenul de atestare
2 # a competentelor profesionale a claselor de matematica-informatica si
3 # matematica-informatica intensiv informatica
4 # Programul in pseudocod nu va face citire din fisier,
5 # deoarece nu am implementat citire din fisier in pseudocod
6 #
7 # daca se citeste de la tastatura
8 # 9
9 # 19 25 5632 9872 48903 33 17634 90 3452
10 # programul va afisa
11 # 48903 17634 9872 90 19
12 subprogramul p_cifra(y) face
13 |   daca y <= 0 atunci
14 |   |   returneaza y
15 |   returneaza p_cifra(floor(y / 10));
16
17 subprogramul sortare(v, n) face
18 |   pentru i <- 1, i <= n, i <- i + 1 executa
19 |   |   pentru j <- i + 1, j <= n, j <- j + 1 executa
20 |   |   |   daca v[i] < v[j] atunci
21 |   |   |   |   aux <- v[i]
22 |   |   |   |   v[i] <- v[j]
23 |   |   |   |   v[j] <- aux
24 |   |
25 ok <- 0
26
27 citeste n
28 v <- vector(n + 1)
29
30 pentru i <- 1, i <= n, i <- i + 1 executa
31 |   citeste x
32 |   cif <- p_cifra(x)
33 |   daca cif = 0 sau cif = 1 sau cif = 4 sau cif = 9 atunci
34 |   |   v[ok+1] <- x
35 |   |   ok <- ok + 1
36
37 daca ok atunci
38 |   sortare(v, ok)
39 |   pentru i <- 1, i <= ok, i <- i + 1 executa
40 |   |   scrie v[i],
41 altfel
42 |   scrie "nu exista"

```

Fisierul trebuie plasat in `~/ .vim/syntax` sau in alt folder echivalent.

Cuprins

Introducere	1
Prezentarea implementarii interpretorului de pseudocod	1
Etapa de lexare/tokenizare	1
Etapa de parsare	1
Etapa de executare	1
Apeluri de functii	1
Variabile	1
Etapa de garbage collection:	2
Implementarea hash mapului	2
Compilare si utilizare a interpretorului	2
Compilare	2
Utilizare	2
Sintaxa utilizata	2
Indentare	2
Expresiile (instructiunile aritmetice)	3
Ordinea operatorilor	3
Variabilele	3
Tipurile de variabile	3
Declararea si apelarea functiilor	3
Functii predefinite	4
Instructiunea de decizie	4
Instructiuni repetitive cu numar nedeterminat de pasi	4
Instructiuni repetitive cu test initial	4
Instructiunea cat timp ... executa	5
Instructiunea pana cand ... executa	5
Instructiuni repetitive cu test final	5
Instructiunea repeta ... cat timp	5
Instructiunea repeta ... pana cand	5
Instructiunea repetitiva cu numar determinat de pasi	5
Instructiunea pentru ... executa	5
Instructiuni de intrare/iesire	5
Instructiuni de intrare	5
Instructiunea citestetot	5
Instructiunea citeste	5
Instructiuni de iesire	6
Instructiunea scrie	6
Vectori	6
Declarea vectoriilor	6
Indexarea vectoriilor	6
Cuvatul cheie sursa	7
Comentarii	7
Exemple de cod in directiva "examples"	7
X si 0	7
Sah	7
Probleme de eficienta	8
Pentru utilizatorii de linux	8
Pentru utilizatorii de vim	8
Cuprins	9