Limbajul Algoritmic

Ștefan Ciobâcă, Dorel Lucanu

Faculty of Computer Science
Alexandru Ioan Cuza University, Iași, Romania
stefan.ciobaca@info.uaic.ro, dlucanu@info.uaic.ro

PA 2016/2017

Plan

Ce este un algoritm?

Cambridge Dictionary:

"A set of mathematical instructions that must be followed in a fixed order, and that, especially if given to a computer, will help to calculate an answer to a mathematical problem."

Schneider and Gersting 1995 (Invitation for Computer Science):

"An algorithm is a well-ordered collection of unambiguous and effectively computable operations that when executed produces a result and halts in a finite amount of time."

Gersting and Schneider 2012 (Invitation for Computer Science, 6nd edition):

"An algorithm is ordered sequence of instructions that is guaranteed to solve a specific problem."

Ce este un algoritm?

Wikipedia:

"In mathematics and computer science, an algorithm is a step-by-step procedure for calculations. Algorithms are used for calculation, data processing, and automated reasoning.

An algorithm is an effective method expressed as a finite list of well-defined instructions for calculating a function. Starting from an initial state and initial input (perhaps empty), the instructions describe a computation that, when executed, proceeds through a finite number of well-defined successive states, eventually producing "output" and terminating at a final ending state. The transition from one state to the next is not necessarily deterministic; some algorithms, known as randomized algorithms, incorporate random input."

Ingredientele de bază: model de calcul, problemă rezolvată

Toate aceste definiții au ceva în comun:

- un model de calcul format din:
  - memorie
  - instrucțiuni
    - sintaxă
    - semantică
- trebuie să rezolve o problemă
  Formalizarea noțiunii de problemă: o pereche (input,output)

Cum descriem un algoritm?

Există o varietate largă de moduri în care poate fi descris un algoritm:

- informal: limbaj natural
- formal
  - notație matematică
  - limbaje de programare
- semiformal
  - pseudo-cod
  - notație grafică

De ce este nevoie de formalizare?

- înainte de secolul 20 matematicienii au utilizat noțiunea de algoritm doar la nivel intuitiv
- în 1900, David Hilbert, la Congresul matematicienilor din Paris, a formulat 23 de probleme ca "provocări ale secolului"
- problema a 10-a cerea "găsirea unui proces care să determine dacă un polinom cu coeficienți întregi are o rădăcină întreagă"
- Hilbert nu a pronunțat termenul de algoritm

De ce nevoie de formalizare?

- problema a 10-a a lui Hilbert este nerezolvabilă
- acest fapt nu se poate demonstra având doar noțiunea intuitivă de algoritm
- pentru a demonstra că nu există algoritm care rezolvă o problemă, este necesară o noțiune formală

Formalizarea noțiunii de algoritm

- Alonso Church, 1936: λ-calcul
- Alan Turing, 1936: "mașini Turing"
- cele două modele de calcul sunt echivalente
- în 1970 Yuri Matijasevic a arătat că problema a 10-a a lui Hibert este nerezolvabilă
- de atunci au apărut multe alte modele echivalente cu mașinile Turing

Teza Church-Turing

Teza lui Turing:
⟨⟨LCMs [logical computing machines: Turing's expression for Turing machines] can do anything that could be described as "rule of thumb" or "purely mechanical".⟩⟩ (Turing 1948)

Teza lui Church:
⟨⟨A function of positive integers is effectively calculable only if recursive.⟩⟩

Kleene (1967) este cel care a introdus termenul de Teza Church-Turing:
"So Turing's and Church's theses are equivalent. We shall usually refer to them both as Church's thesis, or in connection with that one of its . . . versions which deals with Turing machines as the Church-Turing thesis."

Nivelul de formalizare

Care este cel mai potrivit limbaj formal de prezentare a algoritmilor?

- mașinile Turing, lambda-calculul, funcțiile recursive ușor de definit matematic, dar greu de utilizat
- limbajele de programare ușor de utilizat în practică dar dificil de manevrat în demonstrații
- cel mai simplu limbaj de programare echivalent cu mașinile Turing: counting machines
- o variantă structurată : programele `while`

Plan

Motivație

Scopul nostru este de a avea un limbaj care este

- destul de simplu pentru a fi înțeles;
- suficient de expresiv;
- abstract;
- să ofere un model de calcul riguros definit și potrivit pentru analiza algoritmilor;
- executabil;
- intrările și ieșirile sunt reprezentate abstract, ca obiecte matematice.

Un candidat care satisface aceste cerințe este Alk (limbaj specific acestui curs).

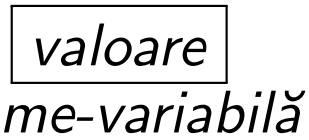
Plan

Modelul de memorie

- memoria este dată de o mulțime de variabile
- o variabilă este o pereche:

  notație matematică $\quad$ *nume-variabilă* $\mapsto$ *valoare*

  notație grafică

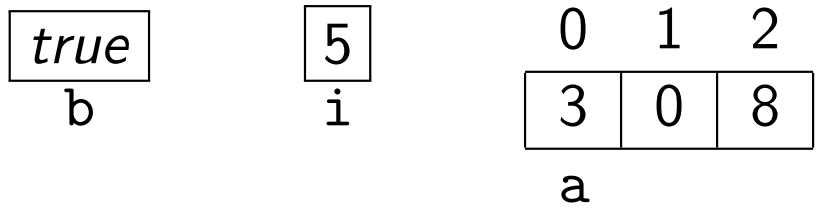
  

- o valoare va fi o constantă a unui tip de date
- exemple de tipuri de valori:
  scalare
  tablouri
  structuri
  liste
  ...
- *Val* desemnează mulțimea tuturor valorilor

Exemple de variabile

notație matematică $\quad \mathtt{b} \mapsto \mathit{true} \quad \mathtt{i} \mapsto 5 \quad a \mapsto [3, 0, 8]$

notație grafică



Fiecare notație este scrierea simplificată (fără numele intermediar al funcției) al unei funcții $\sigma : \{\mathtt{b}, \mathtt{i}, \ldots\} \to \mathit{Val}$ dată prin $\sigma(\mathtt{b}) = \mathit{true}$, $\sigma(\mathtt{i}) = 5$, ...

Plan

Dimensiunea valorilor

Tip de dată = valori (constante) + operații

Fiecare valoare (constantă) este reprezentată utilizând un spațiu de memorie.

Pentru valorile (obiectele) fiecărui tip trebuie precizată lungimea (dimensiunea) reprezentării obiectelor.

Există două moduri de a defini lungimea reprezentării:

- uniform: $|v|_{\text{unif}}$
- logaritmic: $|v|_{\log}$

Scalari

tipurile primitive: valorile boolene, numerele întregi, numerele raționale (virgulă mobilă), șiruri,...

O caracteristică importantă pentru valorile scalare: admit reprezentări finite.

Întrebare: ce se poate spune despre numerele iraționale, de exemplu $\sqrt{2}$?

Scalari (cont)

- întregi:
  $Int = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
  - dimensiunea uniformă: $|n|_{\text{unif}} = 1$
  - dimensiunea logaritmică: $|n|_{\log} = \log_2 n$
- booleeni:
  $Bool = \{false, true\}$
  - dimensiunea uniformă: $|b|_{\text{unif}} = 1$
  - dimensiunea logaritmică: $|b|_{\log} = 1$
- numere raționale:
  $Float =$ numerele raționale
  - dimensiunea uniformă: $|v|_{\text{unif}} = 1$
  - dimensiunea logaritmică: $|v|_{\log} = \log_2(\text{mantisă}) + \log_2(\text{exponent})$
- ...

Avem $Int \cup Bool \cup Float \cup \ldots \subseteq Val$.

Tablouri (unidimensionale)

- $a = [a_0, a_1, \ldots, a_{n-1}]$
- $|a|_d = |a_0|_d + |a_1|_d + \cdots + |a_{n-1}|_d$, $d \in \{\text{unif}, \log\}$
- $Arr_n\langle V\rangle = \{\{0 \mapsto v_0, \ldots, n-1 \mapsto v_{n-1} \mid v_i \in V, i = 0, \ldots, n-1\}$
- $\bigcup_{n\geq 1} Arr_n\langle V\rangle \subset Val$ pentru fiecare tip $V \subset Val$
- tablourile bidimensionale sunt tablouri unidimensionale de tablouri unidimensionale,
- tablourile tridimensionale sunt tablouri unidimensionale de tablouri bidimensionale,
- etc.

Structuri

Exemplu: punctul de coordonate $(2, 7)$ este reprezentat de structura
$$\{\mathtt{x} \to 2\ \mathtt{y} \to 7\}.$$

mulțimea de câmpuri: $F = \{f_1, \ldots, f_n\}$
$s = \{f_1 \to v_1, \ldots, f_n \to v_n\}$

$|s|_d = |v_0|_d + |v_1|_d + \cdots + |v_{n-1}|_d$, $d \in \{\text{unif}, \log\}$

$Str\langle f_1{:}V_1, \ldots f_n{:}V_n\rangle = \{\{f_1 \to v_1, \ldots, f_n \to v_n\} \mid v_1 \in V_1, \ldots, f_n \in V_n\}$

Exemplu (Fixed Size Linear Lists):
$FSLL = \{\mathtt{len}, \mathtt{arr}\}$
$Str\langle \mathtt{len} : Int, \mathtt{arr} : Arr_{100}\langle Int\rangle\rangle =$
$\{\{\mathtt{len} \to n\ \mathtt{arr} \to a\} \mid n \in Int, a \in Arr_{100}\langle Int\rangle\}$

$Str\langle f_1{:}V_1, \ldots f_n{:}V_n\rangle \subset Val$ pentru fiecare structură $F = \{f_1{:}V_1, \ldots f_n{:}V_n\}$.

Liste liniare

O valoare de tip listă liniară este o secvență de valori $l = \langle v_0, v_1, \ldots, v_{n-1} \rangle$.

$|l|_d = |v_0|_d + |v_1|_d + \cdots + |v_{n-1}|_d$, $d \in \{\text{unif}, \log\}$

$LLin\langle V \rangle = \{\langle v_0, \ldots, v_{n-1} \rangle \mid v_i \in V, i = 0, \ldots, n\}$

Exemple: $LLin\langle Int \rangle$, $LLin\langle Arr_n \rangle$, $LLin\langle Arr_n\langle Float \rangle\rangle$

Avem $LLin\langle V \rangle \subset Val$ pentru fiecare tip V.

Valori complexe: grafuri

Graful $G = (\{0, 1, 2, 3\}, \{(0, 1), (0, 2), (0, 3), (1, 2)\})$ este reprezentat prin liste de adiacență de următoarea valoare:

$$\{$$
$$\quad \mathtt{n} \to 4$$
$$\quad \mathtt{a} \to [\langle 1, 2, 3\rangle, \langle 0, 2\rangle, \langle 0, 1\rangle, \langle 0\rangle]$$
$$\}$$

Plan

Tip de date (cont.)

Tip de dată = constante + operații

Fiecare operație *op* are un cost timp $time(op)$.

Pentru fiecare operație a unui tip trebuie precizat timpul.

Există două moduri de a defini timpul (moștenite de la lungimea reprezentării):
uniform: $time_{\text{unif}}(op)$ – utilizează dimensiunea uniformă
logaritmic: $time_{\log}(op)$ – utilizează dimensiunea logaritmică

Operații cu scalari

Întregi:

Operație	$time_{\text{unif}}(op)$	$time_{\log}(op)$
$a +_{Int} b$	$O(1)$	$O(\max(\log a, \log b))$
$a *_{Int} b$	$O(1)$	$O(\log a \cdot \log b)$ $O(\max(\log a, \log b)^{1.545})$
...	...	...

Tablouri

Operație	$time_{\text{unif}}(op)$	$time_{\log}(op)$
$A.\texttt{lookup}(i)$	$O(1)$	$O(i + \log a_i)$
$A.\texttt{update}(i, v)$	$O(1)$	$O(i + \log v)$

unde am presupus $A \mapsto [a_0, \ldots, a_{n-1}]$

Structuri

Operație	$time_{\text{unif}}(op)$	$time_{\log}(op)$
$S.\texttt{lookup}(x)$	$O(1)$	$O(\log s_x)$
$S.\texttt{update}(x, v)$	$O(1)$	$O(\log v)$

unde am presupus $S \mapsto \{\ldots x \to s_x, \ldots\}$

Liste liniare: definiție operații

`empty()`	întoarce lista vidă []
L`.topFront()`	întoarce v_0
L`.topBack()`	întoarce v_{n-1}
L`.lookup(`i`)`	întoarce v_i
L`.insert(`i`,`x`)`	întoarce $[\ldots v_{i-1}, x, v_i, \ldots]$
L`.remove(`i`,`x`)`	întoarce $[\ldots v_{i-1}, v_{i+1}, \ldots]$
L`.size()`	întoarce n
L`.popFront()`	întoarce $[v_1, \ldots, v_{n-1}]$
L`.popBack()`	întoarce $[v_0, \ldots, v_{n-2}]$
L`.pushFront(`x`)`	întoarce $[x, v_0, \ldots, v_{n-1}]$
L`.pushBack(`x`)`	întoarce $[v_0, \ldots, v_{n-1}, x]$
L`.update(`i`,`x`)`	întoarce $[\ldots v_{i-1}, x, v_{i+1}, \ldots]$

unde am presupus $L \mapsto [v_0, \ldots, v_{n-1}]$

Liste liniare: operații (versiunea 1)

- corespunde implementării cu tablouri

Operație	$time_{\text{unif}}(op)$	$time_{\log}(op)$
L.lookup(i)	$O(1)$	?
L.insert(i, x)	$O(L.\text{size}() - i)$	?
L.remove(i, x)	$O(L.\text{size}() - i)$	?
L.update(i, x)	$O(1)$	?
L.topFront()	$O(1)$	?
L.popFront()	$O(L.\text{size}())$	?
L.pushFront()	$O(L.\text{size}()))$	?
L.topBack()	$O(1)$	?
L.popBack()	$O(1)$	?
L.pushBack()	$O(1)$	?

Liste liniare: operații (versiunea 2)

- corespunde implementării cu liste dublu înlănțuite

Operație	$time_{\text{unif}}(op)$	$time_{\log}(op)$
L.lookup(i)	$O(i)$	?
L.insert(i, x)	$O(i)$	?
L.remove(i, x)	$O(i)$	?
L.update(i, x)	$O(i)$	?
L.topFront()	$O(1)$	?
L.popFront()	$O(1)$	?
L.pushFront()	$O(1)$	?
L.topBack()	$O(1)$	?
L.popBack()	$O(1)$	?
L.pushBack()	$O(1)$	?

Liste liniare: operații (versiunea 3?)

- Există o listă liniară cu următoarele proprietăți?

Operație	$time_{\text{unif}}(op)$	$time_{\log}(op)$
L.lookup(i)	$O(1)$	?
L.insert(i, x)	$O(i)$	?
L.remove(i, x)	$O(i)$	?
L.update(i, x)	$O(1)$	?
L.topFront()	$O(1)$	?
L.popFront()	$O(1)$	?
L.pushFront()	$O(1)$	?
L.topBack()	$O(1)$	?
L.popBack()	$O(1)$	?
L.pushBack()	$O(1)$	?

Plan

Expresii: sintaxa

Sintaxa este foarte apropiata de cea limbajului C++:

- expresii aritmetice: `a * b + 2`
- expresii relaționale: `a < 5`
- expresii booleene: `(a < 5) && (a > -1)`
- expresii peste mulțimi: `s1 U s2` `s1 ^ s2` `s1 \ s2`
- apel de funcție: `f(a*2, b+5)`
- apel operații peste liste, mulțimi, tablouri:
  `l.update(2,55)` `l.size()`

Instrucțiuni: sintaxa

- atribuire `a =` E`;` `a[i] =` E`;` `p.x =` E`;`
- apeluri de funcții: `quicksort(a);` `l.insert(2,77);`
- bloc: `{` Sts `}`
- instrucțiuni condiționale:
  - `if (` E `)` St
  - `if (` E `)` St_1 `else` St_2
- instrucțiuni repetitive:
  - `while (` E `)` St
  - `forall` X `in` S St
  - `for (` X `=` E`;` E'`; ++`X `)` S
- return: `return` E`;`
- compunerea secvențială: St_1 St_2

Alk este extensibil: pot fi adăugate noi expresii și instrucțiuni, cu precizări pentru costul timp

Tipuri de date

Sunt predefinite în Alk.

Presupunem existența unei metainformații care menționează tipul fiecărei variabile (nu există declarații de variabile).

Exemplu de program

```
/*
  This example includes the recursive version of the DFS algorithm.
  @input: a digraf D and a vertex i0
  @output: the list S of the vertices reachable from i0
*/

 // the recursive function
 dfsRec(i) {
   if (S[i] == 0) {
     // visit i
     S[i] = 1;
     p = D.a[i];
     while (p.size() > 0) {
       j = p.topFront();
       p.popFront();
       dfsRec(j);
     }
   }
 }
```

```
// the calling program
i = 0;
while (i < D.n) {
  S[i] = 0;
  i = i + 1;
}
dfsRec(1);
```

Evaluarea expresiilor

Considerăm o funcție $[\![\_]\!](\_) : \mathit{Expresii} \to (\mathit{Stare} \to \mathit{Valori})$, unde $[\![E]\!](\sigma)$ întoarce valoarea expresiei E calculată în starea σ.

Exemplu: Fie σ o stare ce include $\mathtt{a} \mapsto 3 \quad \mathtt{b} \mapsto 6$. Avem:
$[\![\mathtt{a} + \mathtt{b} * 2]\!](\sigma) =$
$[\![\mathtt{a}]\!](\sigma) +_{Int} [\![\mathtt{b} * 2]\!](\sigma) =$
$3 +_{Int} [\![b]\!](\sigma) *_{Int} [\![2]\!](\sigma) =$
$3 +_{Int} 6 *_{Int} 2 =$
$3 +_{Int} 12 = 15$ unde $+_{Int}$ reprezintă algoritmul de adunare peste întregi și $*_{Int}$ reprezintă algoritmul de înmulțire peste întregi.

Calculul timpului pentru evaluare

$time_d([\![\mathtt{a} + \mathtt{b} * 2]\!](\sigma)) =$
$time_{\mathtt{d}}([\![\mathtt{a}]\!](\sigma)) + time_{\mathtt{d}}([\![\mathtt{b}]\!](\sigma)) + time_{\mathtt{d}}(6 *_{Int} 2) + time_{\mathtt{d}}(3 +_{Int} 12)$,
$d \in \{\mathrm{unif}, \log\}$.

σ include $\mathtt{a} \mapsto 3 \quad \mathtt{b} \mapsto 6$
$time_{\log}([\![\mathtt{a}]\!](\sigma)) = \log 3,\ time_{\log}([\![\mathtt{b}]\!](\sigma)) = \log 6$
$time_{\mathrm{unif}}([\![\mathtt{a}]\!](\sigma)) = 1,\ time_{\mathrm{unif}}([\![\mathtt{b}]\!](\sigma)) = 1$

Configurații

O configurație este o pereche $\langle$*secvență-de-program*, *stare*$\rangle$

Exemple:

$\langle$`x = x + 1; y = y + 2 * x;`, $\texttt{x} \mapsto 7\ \texttt{y} \mapsto 12\rangle$

$\langle$`s = 0; while (x > 0) {s = s+x; x = x-1;}`, $\texttt{x} \mapsto 5\ \texttt{s} \mapsto -15\rangle$

Pași de execuție

Un pas de execuție este definit ca o tranziție între configurații:

$$\langle S, \sigma \rangle \Rightarrow \langle S', \sigma' \rangle$$

dacă și numai dacă
executând prima instrucțiune secvența din S în starea σ obținem secvența S', ce urmează a fi executată în continuare, și o nouă stare σ'

Pașii de execuție sunt descriși prin reguli $\langle S_1, \sigma_1 \rangle \Rightarrow \langle S_2, \sigma_2 \rangle$, unde $S_1, S_2, \sigma_1, \sigma_2$ sunt termeni cu variabile (patterns).

Pentru a putea calcula costul timp al unui pas de execuție, vom preciza costul dat de aplicarea unei reguli.

Instrucțiuni: semantica

atribuirea: $\mathtt{x} = E;$

- *informal*: se evaluează E și rezultatul este atribuit ca noua valoare a variabilei $\mathtt{x}$
- *formal*:
  $\langle \mathtt{x} = E; S, \sigma \} \rangle \Rightarrow \langle S, \sigma' \rangle$
  unde σ este de forma $\ldots \mathtt{x} \mapsto v \ldots$ și σ' de forma $\ldots \mathtt{x} \mapsto [\![\mathtt{E}]\!](\sigma) \ldots$
  (în rest la fel ca σ).

Costul timp:
$time_{\text{unif}}(\langle \mathtt{x} = E; S, \sigma \rangle \Rightarrow \langle S, \sigma' \rangle) = time_{\text{unif}}([\![E]\!](\sigma)) + 1,$
$time_{\log}(\langle \mathtt{x} = E; S, \sigma \rangle \Rightarrow \langle S, \sigma' \rangle) = time_{\log}([\![E]\!](\sigma) + \log [\![E]\!](\sigma)).$

Instrucțiuni: semantica

if: `if (E) then S else S'`

- *informal*: se evaluează e; dacă rezultatul obținut este *true*, atunci se execută S, altfel se execută S'
- *formal*:

  $\langle \texttt{if (E) then S else S'}\ \ S'', \sigma\rangle \Rightarrow \langle S\ \ S'', \sigma\rangle$ dacă $[\![E]\!](\sigma) = \mathit{true}$

  $\langle \texttt{if (E) then S else S'}\ \ S'', \sigma\rangle \Rightarrow \langle S'\ \ S'', \sigma\rangle$ dacă $[\![E]\!](\sigma) = \mathit{false}$

Costul timp:
$\mathit{time}_d(\langle \texttt{if (E) then S' else S''}\ \ S, \sigma\rangle \Rightarrow \langle \_, \sigma\rangle) = \mathit{time}_d([\![E]\!](\sigma))$
$d \in \{\mathrm{unif}, \log\}$.

Instrucțiuni: semantica

while: `while (`E`)` S

- *informal*: se evaluează e; dacă rezultatul obținut este *true*, atunci se execută S, după care se evaluează din nou e și ...; altfel execuția instrucțiunii se termină
- *formal*: se exprimă cu ajutorul lui `if`:
  $\langle$`while (`e`)` S $S', \sigma\rangle \Rightarrow$
  $\langle$`if (`e`) { `S `; while (`e`)` S `} else { }`$S', \sigma\rangle$

Costul timp:
$time_d(\langle$`while (`E`) then` S `else` S' $S, \sigma\rangle \Rightarrow \langle$`if (`$e$`) ...`$S, \sigma\rangle) = 0$,
$d \in \{\text{unif}, \log\}$.

Apelul de funcție

Presupunem funcția `f(a,b) { S_f }`. Evaluarea apelului `f`(e_1,e_2) presupune următorii pași:

$\langle \texttt{f}(e_1, e_2)\ S, \sigma, \mathit{Stack}\rangle \Rightarrow$

$\langle S_f, \sigma \cup \{\texttt{a} \mapsto [\![e_1]\!](\sigma)\ \texttt{b} \mapsto [\![e_2]\!](\sigma)\}, (S, \sigma)\ \mathit{Stack}\rangle \Rightarrow^*$

$\langle v, \sigma', (S, \sigma)\ \mathit{Stack}\rangle \Rightarrow$

$\langle v\ S, \mathit{updateGlobals}(\sigma, \sigma'), \mathit{Stack}\rangle$

Presupunere: costul unui apel este suma dintre costul evaluării argumentelor actuale și costul execuției corpului funcției.

Calcul (execuție)

Un calcul (o execuție) este o secvență de pași:
$\tau = \langle S_1, \sigma_1 \rangle \Rightarrow \langle S_2, \sigma_2 \rangle \Rightarrow \langle S_3, \sigma_3 \rangle \Rightarrow \ldots$

Costul unui calcul:
$time_d(\tau) = \sum_i time_d(\langle S_i, \sigma_i \rangle \Rightarrow \langle S_{i+1}, \sigma_{i+1} \rangle),\ d \in \{unif, log\}$

Calcul: exemplu

$\langle$`if (x > 3) x = x + y; else x = 0; y = 4;` $, x \mapsto 7\ y \mapsto 12\rangle \Rightarrow$
$\langle$`x = x + y; y = 4;` $, x \mapsto 7\ y \mapsto 12\rangle \Rightarrow$
$\langle$`y = 4;` $, x \mapsto 19\ y \mapsto 12\rangle \Rightarrow$
$\langle \cdot, x \mapsto 19\ y \mapsto 4\rangle$

Am utilizat:
$[\![x > 3]\!](x \mapsto 7\ y \mapsto 12) = \mathit{true}$
$[\![x + y]\!](x \mapsto 7\ y \mapsto 12) = 19$
$[\![4]\!](x \mapsto 19\ y \mapsto 12) = 4$

Costul:
cost uniform: 3 (= numărul de pași)
cost logaritmic: $\log 7 + \log 7 + \log 12 + \log 19 + \log 4$

Calcul: exemplu

$\langle \texttt{while (i > 5) i--;}\ , \texttt{i} \mapsto 6\ \texttt{x} \mapsto 12\rangle \Rightarrow$
$\langle \texttt{if (i > 5) \{ i --; while (i > 5) i--; \}}, \texttt{i} \mapsto 1\ \texttt{x} \mapsto 12\rangle \Rightarrow$
$\langle \texttt{\{i --; while (i > 5) i--; \}}\ , \texttt{i} \mapsto 6\ \texttt{x} \mapsto 12\rangle \Rightarrow$
$\langle \texttt{i --; while (i > 5) i--;}\ , \texttt{i} \mapsto 6\ \texttt{x} \mapsto 12\rangle \Rightarrow$
$\langle \texttt{while (i > 5) i--;}\ , \texttt{i} \mapsto 5\ \texttt{x} \mapsto 12\rangle \Rightarrow$
$\langle \cdot, \texttt{i} \mapsto 5\ \texttt{x} \mapsto 12\rangle$

Am utilizat:
$[\![\texttt{i} > 5]\!](\texttt{i} \mapsto 6\ \texttt{x} \mapsto 12) = \mathit{true}$
$[\![\texttt{i} - -]\!](\texttt{i} \mapsto 6\ \texttt{x} \mapsto 12) = 0$
$[\![\texttt{i} > 5]\!](\texttt{i} \mapsto 5\ \texttt{x} \mapsto 12) = \texttt{false}$

Costul timp:
cost uniform: 4 (= numărul de pași, exceptând primul)
cost logaritmic: $\log 6 + \log 6 + \log 5 + \log 5$

Plan

Testarea algoritmilor cu K Framework

K Framework (www.kframework.org) este un mediu de lucru pentru definiții de limbaje de programare. Definițiile K sunt executabile.

Definiția K a lui Alk se găsește la adresa
https://github.com/kframework/alk-semantics
și poate fi compilată cu K versiunea 3.6:
https://github.com/kframework/k/releases
O copie actualizată va fi accesibilă de pe pagina cursului.

Compilarea definiției limbajului Alk:
kompile alk

Proiecte (posibil pentru licență):
1. implementare C++ a definiției lui Alk;
2. dezvoltarea unei interfețe specifice pentru definiția lui Alk.

Execuția algoritmului DFS recursiv

Starea inițială este dată ca valoare a opțiunii `init`:

```
alki dfsrec.alk \
     --init="D |-> { n -> 3
                     a -> [ < 1, 2 >, < 2, 0 >,  < 0 > ] }
               i0 |-> 1"
```

Obținerea configurației finale

Configurația finală obținută prin comanda precedentă este:

```
State:

    D |-> { (n -> 3) (a -> ([ (< 1, 2 >), (< 2, 0 >), (< 0 >) ])) }
    i0 |-> 1
    reached |-> [ 1, 1, 1 ]
```

Demo

Execuția algoritmilor de mai sus.