

# Limbajul Algoritmico

Şt. Ciobâcă, Dorel Lucanu

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

PA 2015/2016

## 1 Introducere

## 2 Limbajul Alk

- Modelul de memorie
- Valori
- Operații
- Expresii și instrucțiuni
  - Sintaxa
  - Semantica

## 3 Testarea algoritmilor cu K Framework

# Plan

## 1 Introducere

## 2 Limbajul Alk

- Modelul de memorie
- Valori
- Operații
- Expresii și instrucțiuni
  - Sintaxa
  - Semantica

## 3 Testarea algoritmilor cu K Framework

# 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:  $\lambda$ -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 Hilbert 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

## 1 Introducere

## 2 Limbajul Alk

- Modelul de memorie
- Valori
- Operații
- Expresii și instrucțiuni
  - Sintaxa
  - Semantica

## 3 Testarea algoritmilor cu K Framework

# 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

## 1 Introducere

## 2 Limbajul Alk

- Modelul de memorie
- Valori
- Operații
- Expresii și instrucțiuni
  - Sintaxa
  - Semantica

## 3 Testarea algoritmilor cu K Framework

# Modelul de memorie

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

notație matematică     $\text{nume-variabilă} \mapsto \text{valoare}$

notație grafică

valoare

  
nume-variabilă

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



# Exemple de variabile

notație matematică  $b \mapsto true$     $i \mapsto 5$     $a \mapsto [3, 0, 8]$

notație grafică

<i>true</i>
-------------

*b*

5
---

*i*

0	1	2
3	0	8

*a*

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

# Plan

## 1 Introducere

## 2 Limbajul Alk

- Modelul de memorie
- **Valori**
- Operații
- Expresii și instrucțiuni
  - Sintaxa
  - Semantica

## 3 Testarea algoritmilor cu K Framework

# Dimensiunea valorilor

Tip de dată = valori (obiecte) + operații

Fiecare valoare (obiect) 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:  $|v|_{\text{log}}$

# 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:

$$Int = \{\dots, -2, -1, 0, 1, 2, \dots\}$$

- 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:

$$Float = \text{numerele raționale}$$

- dimensiunea uniformă:  $|v|_{\text{unif}} = 1$
- dimensiunea logaritmică:  $|v|_{\log} = \log_2(\text{mantisă}) + \log_2(\text{exponent})$

- ...

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

# Tablouri (unidimensionale)

- $a = [a_0, a_1, \dots, a_{n-1}]$
- $|a|_d = |a_0|_d + |a_1|_d + \dots + |a_{n-1}|_d, d \in \{\text{unif}, \log\}$
- $Arr_n\langle V \rangle = \{\{0 \mapsto v_0, \dots, n-1 \mapsto v_{n-1} \mid v_i \in V, i = 0, \dots, n-1\}$
- $\bigcup_{n \geq 1} Arr_n\langle V \rangle \subset Val$  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  
 $\{x \rightarrow 2 \ y \rightarrow 7\}.$

$$F = \{f_1, \dots, f_n\}$$

$$s = \{f_1 \rightarrow v_1, \dots, f_n \rightarrow v_n\}$$

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

$$\text{Str}\langle f_1:V_1, \dots f_n:V_n \rangle = \{\{f_1 \rightarrow v_1, \dots, f_n \rightarrow v_n\} \mid v_1 \in V_1, \dots, f_n \in V_n\}$$

Exemplu (Fixed Size Linear Lists):

$$FSLL = \{\text{len}, \text{arr}\}$$

$$\text{Str}\langle \text{len} : \text{Int}, \text{arr} : \text{Arr}_{100}\langle \text{Int} \rangle \rangle =$$

$$\{\{\text{len} \rightarrow n \ \text{arr} \rightarrow a\} \mid n \in \text{Int}, a \in \text{Arr}_{100}\langle \text{Int} \rangle\}$$

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

# Liste liniare

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

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

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

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:

$$\begin{aligned} &\{ \\ &\quad n \rightarrow 4 \\ &\quad a \rightarrow [\langle 1, 2, 3 \rangle, \langle 0, 2 \rangle, \langle 0, 1 \rangle, \langle 0 \rangle] \\ &\} \end{aligned}$$

# Plan

## 1 Introducere

## 2 Limbajul Alk

- Modelul de memorie
- Valori
- **Operații**
- Expresii și instrucțiuni
  - Sintaxa
  - Semantica

## 3 Testarea algoritmilor cu K Framework

## Tip de date (cont.)

Tip de dată = obiecte + 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:**  $time_{unif}(op)$  – utilizează dimensiunea uniformă

**logaritm:**  $time_{log}(op)$  – utilizează dimensiunea logaritmă

# Operații cu scalari

Întregi:

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

# Tablouri

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

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

# Structuri

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

unde am presupus  $S \mapsto \{\dots x \rightarrow s_x, \dots\}$

# Liste liniare: definiție operații

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

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

# Liste liniare: operații (versiunea 1)

- corespunde implementării cu tablouri

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



# Liste liniare: operații (versiunea 2)

- corespunde implementării cu liste dublu înlănțuite

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

# Liste liniare: operații (versiunea 3?)

- Există o listă liniară cu următoarele proprietăți?

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

# Plan

## 1 Introducere

## 2 Limbajul Alk

- Modelul de memorie
- Valori
- Operații
- Expresii și instrucțiuni
  - Sintaxa
  - Semantica

## 3 Testarea algoritmilor cu K Framework

# Expresii: sintaxa

Sintaxa este foarte apropiată de cea limbajului C++:

- expresii aritmetice:  $a * b + 2$
- expresii relaționale:  $a < 5$
- expresii booleene:  $(a < 5) \ \&\& \ (a > -1)$
- expresii peste mulțimi:  $s1 \cup s2$      $s1 \wedge s2$      $s1 \setminus s2$
- 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:
  - $\text{if } ( E ) St$
  - $\text{if } ( E ) St_1 \text{ else } St_2$
- instrucțiuni repetitive:
  - $\text{while } ( E ) St$
  - $\text{forall } X \text{ in } S St$
  - $\text{for } ( X = E; E'; ++X ) S$
- return:  $\text{return } E$ ;
- 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  $\llbracket \_ \rrbracket (-) : \text{Expresii} \rightarrow (\text{Stare} \rightarrow \text{Valori})$ , unde  $\llbracket E \rrbracket (\sigma)$  întoarce valoarea expresiei  $E$  calculată în starea  $\sigma$ .

Exemplu: Fie  $\sigma$  o stare ce include  $a \mapsto 3$   $b \mapsto 6$ . Avem:

$$\llbracket a + b * 2 \rrbracket (\sigma) =$$

$$\llbracket a \rrbracket (\sigma) +_{Int} \llbracket b * 2 \rrbracket (\sigma) =$$

$$3 +_{Int} \llbracket b \rrbracket (\sigma) *_{Int} \llbracket 2 \rrbracket (\sigma) =$$

$$3 +_{Int} 6 *_{Int} 2 =$$

$3 +_{Int} 12 = 15$  unde  $+_{Int}$  reprezintă algoritmul de adunare peste întregi și  $*_{Int}$  reprezintă algoritmul de înmulțire peste întregi.



# Calculul timpului pentru evaluare

$$\begin{aligned} time_d(\llbracket a + b * 2 \rrbracket(\sigma)) &= \\ time_d(\llbracket a \rrbracket(\sigma)) + time_d(\llbracket b \rrbracket(\sigma)) + time_d(6 *_{Int} 2) + time_d(3 +_{Int} 122), \\ d &\in \{\text{unif}, \log\}. \\ time_{\log}(\llbracket a \rrbracket(\sigma)) &= \log 3, \quad time_{\log}(\llbracket b \rrbracket(\sigma)) = \log 6 \\ time_{\text{unif}}(\llbracket a \rrbracket(\sigma)) &= 1, \quad time_{\text{unif}}(\llbracket b \rrbracket(\sigma)) = 1 \end{aligned}$$

# Configurații

O configurație este o pereche  $\langle \text{secvență-de-program}, \text{stare} \rangle$

Exemple:

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

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

# 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  $\sigma$  obținem secvența  $S'$ , ce urmează a fi executată în continuare, și o nouă stare  $\sigma'$

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:  $x = E;$

- *informal*: se evaluează  $E$  și rezultatul este atribuit ca noua valoare a variabilei  $x$

- *formal*:

$$\langle x = E; S, \sigma \rangle \Rightarrow \langle S, \sigma' \rangle$$

unde  $\sigma$  este de forma  $\dots x \mapsto v \dots$  și  $\sigma'$  de forma  $\dots x \mapsto \llbracket E \rrbracket(\sigma) \dots$  (în rest la fel ca  $\sigma$ ).

Costul timp:

$$time_{\text{unif}}(\langle x = E; S, \sigma \rangle \Rightarrow \langle S, \sigma' \rangle) = time_{\text{unif}}(\llbracket E \rrbracket(\sigma)) + 1,$$

$$time_{\text{log}}(\langle x = E; S, \sigma \rangle \Rightarrow \langle S, \sigma' \rangle) = time_{\text{log}}(\llbracket E \rrbracket(\sigma)) + \log \llbracket E \rrbracket(\sigma).$$

# 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*:

$\langle \text{if } (E) \text{ then } S \text{ else } S' \ S'', \sigma \rangle \Rightarrow \langle S \ S'', \sigma \rangle$  dacă  $\llbracket E \rrbracket(\sigma) = \text{true}$

$\langle \text{if } (E) \text{ then } S \text{ else } S' \ S'', \sigma \rangle \Rightarrow \langle S' \ S'', \sigma \rangle$  dacă  $\llbracket E \rrbracket(\sigma) = \text{false}$

Costul timp:

$\text{time}_d(\langle \text{if } (E) \text{ then } S' \text{ else } S'' \ S, \sigma \rangle \Rightarrow \langle -, \sigma \rangle) = \text{time}_d(\llbracket E \rrbracket(\sigma))$

$d \in \{\text{unif}, \text{log}\}.$

# 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*:  

$$\langle \text{while } (e) \ S \ S', \sigma \rangle \Rightarrow$$

$$\langle \text{if } (e) \ \{ \ S \ ; \ \text{while } (e) \ S \} \ \text{else } \{ \} S', \sigma \rangle$$

Costul timp:

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

# Apelul de funcție

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

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

$$\langle S_f, \sigma \cup \{a \mapsto \llbracket e_1 \rrbracket(\sigma) \ b \mapsto \llbracket e_2 \rrbracket(\sigma)\}, (S, \sigma) \ \text{Stack} \rangle \Rightarrow^*$$

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

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

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:

$$\tau = \langle S_1, \sigma_1 \rangle \Rightarrow \langle S_2, \sigma_2 \rangle \Rightarrow \langle S_3, \sigma_3 \rangle \Rightarrow \dots$$

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

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

Am utilizat:

$\llbracket x > 3 \rrbracket (x \mapsto 7 \ y \mapsto 12) = \text{true}$

$\llbracket x + y \rrbracket (x \mapsto 7 \ y \mapsto 12) = 19$

$\llbracket 4 \rrbracket (x \mapsto 19 \ y \mapsto 12) = 4$

Costul:

cost uniform: 3 (= numărul de pași)

cost logaritmice:  $\log 7 + \log 7 + \log 12 + \log 19 + \log 4$

## Calcul: exemplu

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

Am utilizat:

$\llbracket i > 5 \rrbracket (i \mapsto 6 \ x \mapsto 12) = \text{true}$

$\llbracket i-- \rrbracket (i \mapsto 6 \ x \mapsto 12) = 0$

$\llbracket i > 5 \rrbracket (i \mapsto 5 \ x \mapsto 12) = \text{false}$

Costul timp:

cost uniform: 4 (= numărul de pași, exceptând primul)

cost logaritmice:  $\log 6 + \log 6 + \log 5 + \log 5$

# Plan

- 1 Introducere
- 2 Limbajul Alk
  - Modelul de memorie
  - Valori
  - Operații
  - Expresii și instrucțiuni
    - Sintaxa
    - Semantica
- 3 Testarea algoritmilor cu K Framework

# Testarea algoritmilor cu K Framework

K Framework ([www.kframework.org](http://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**

**Proiect (posibil pentru licență):** dezvoltarea unei interfețe specifice pentru Alk.

# Execuția algoritmului DFS recursiv

Starea inițială este dată ca valoare a opțiunii cINIT:

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

# Obținerea configurației finale

Configurația finală obținută prin comanda precedentă este:

```
<k>
.K
</k>
<state>
D |-> { n -> 3 a -> ([ (< 1, 2 >), (< 2, 0 >), (< 0, .ValueList >) ]) }
S |-> [ 1, 1, 1 ]
dfsRec |-> lambda ( i ) . ({ if ( S [ i ] == 0 ) { ((S [ i ] = 1) ;) ((
    = (D . a) [ i ]) ;) (while ( p . size ( ) > 0 ) { ((j = p . topFront
    ( ) ;) (p . popFront ( ) ;) (dfsRec ( j ) ;) }) } })
i0 |-> 1
i |-> 3
</state>
```

# Demo

Execuția algoritmilor de mai sus.