Algorithmic Language

Ștefan Ciobâcă, Dorel Lucanu

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

PA 2019/2020

1. Introduction

2. Alk Language
   - Memory model
   - Values
   - Operations
   - Expressions and instructions
     - Syntax
     - Semantics

3. Testing the algorithms with Alk Interpreter

Plan

What is an algorithm?

What is an algorithm?

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."

What is an algorithm?

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."

What is an algorithm?

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

What is an algorithm?

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.

What is an algorithm?

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.

What is an algorithm?

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.

What is an algorithm?

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."

Basic ingredients: computation model, solved problem

All these definitions share the following items:

- a computation model consisting of:
  - memory/store/state/configuration
  - instructions/commands

    syntax

    semantics
- an algorithm must solve a problem

Basic ingredients: computation model, solved problem

All these definitions share the following items:

- a computation model consisting of:
  - memory/store/state/configuration
  - instructions/commands

    syntax

    semantics
- an algorithm must solve a problem

  A definition for problem in this context: a pair (input,output)

How to describe an algorithm?

There are various ways to describe an algorithm:

- informal: natural language
- formal
    - mathematical notation
    - programming languages
- semiformal
    - pseudo-code
    - graphical notation

Is formalisation needed?

Is formalisation needed?

- before the 20th century only intuitive definitions for algorithm were used
- in 1900, at the Congress of the mathematicians from Paris, David Hilbert formulated 23 problems as "challenges of the new century"
- the 10th problem asked for "finding a process that determines whether an integer polynomial has an integer root"
- Hilbert didn't pronounce the term of algorithm

Is formalisation needed?

- Hibert's 10th problem is non-solvable/non-computable
- this fact cannot be proved having only the intuitive notion of algorithm
- to prove that there is no algorithm that solve this problem, we need a formal definition for algorithm

Concept of algorithm, formally

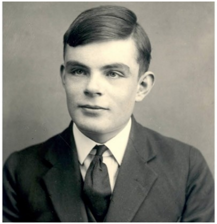
- 1933, Kurt Gdel, with Jacques Herbrand: general (partial) recursive functions
  Alonso Church, 1936: λ-calculus
- Alan Turing, 1936: Turing machines
- the three models are equivalent
- since then were defined many other computation models equivalent to Turing machines
- în 1970 Yuri Matijasevic showed that the Hibert's 10th problem is non-solvable

λ-calculus

- The language:
  x
  $\lambda x.M$ (abstraction)
  $M\ N$ (application)
- Booleans:
  $true \triangleq \lambda a.\lambda b.a$
  $false \triangleq \lambda a.\lambda b.b$
- Integers:
  $0 \triangleq \lambda f.\lambda x.x$ (equivalent to *false*)
  $1 \triangleq \lambda f.\lambda x.f\ x$
  $2 \triangleq \lambda f.\lambda x.f(f\ x)$
  ...
  $succ = \lambda n.\lambda f.\lambda x.f(n\ f\ x)$
- Operational Semantics:
  $(\lambda x.M)N \Rightarrow M[N/x]$ (β-reduction)

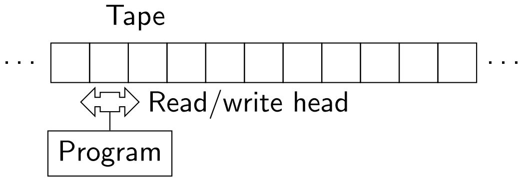
Sursa: https://en.wikipedia.org/wiki/Alonzo_Church

Turing Machine 1/3



Source: https://www.iwm.org.uk/history/how-alan-turing-cracked-the-enigma-code
https://www.decodedscience.org/what-is-universal-turing-machine/12081

Turing Machine 2/3

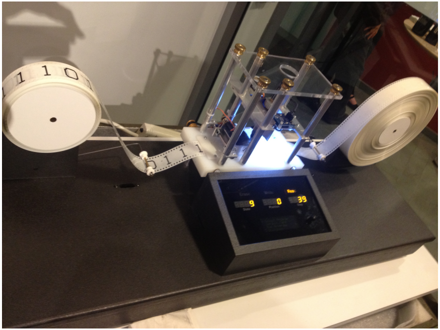


Instruction: $\langle q, s, q', s', d\rangle$, where
$q, q' \in Q$ (= a finite set of states)
$s, s' \in \Sigma$ (= finite alphabet)
$d \in \{L, R, N\}$ (= moving directions)

Turing Machine 3/3

q_4

	s_1	s_1	s_3	s_1	s_0	s_1	



Source:By GabrielF - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=26270095

Church-Turing Thesis

Turing Thesis:
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)

Church Thesis:
Real-world calculation can be done using the lambda calculus, which is equivalent to using general recursive functions. (Church 1935, 1936)

Kleene (1967) introduced the term of Church-Turing Thesis:
"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."

The level of formalisation

What is the most suitable language for representing the algorithms?

- Turing machines, lambda-calcululus, recursive functions: easy mathematical definitions, hard to use in practice
- programming languages easy(?) to use in practice, hard to use in proofs
- the most simple language equivalent to Turing machines: counting machines
- a structured version : `while` programs

Plan

Motivation

The goal (for this lecture) is to have a language that is:

- simple to be easily understood;
- expressive enough;
- abstract;
- to supply a rigurous computation model suitable to analyse algorithms;
- executable;
- input and output are given as abstract data types.

The Alk language was developed to meet these requirements.

Plan

Memory model

- the memory is a set of variables
- a variable is a pair:

mathematical notation $\quad$ *variabile-name* $\mapsto$ *value*

graphical notation $\quad$ $\boxed{value}$ *variabile-name*

- a value is an object of an (abstract) data type
- examples of values:
  scalars
  arrays
  structures
  lists
  ...
- *Val* denotes the set of all values

Examples of variables

math notation $\quad \texttt{b} \mapsto \mathit{true} \quad \texttt{i} \mapsto 5 \quad a \mapsto [3, 0, 8]$

graphical notation $\quad \underset{\texttt{b}}{\boxed{\mathit{true}}} \quad \underset{\texttt{i}}{\boxed{5}}$

0	1	2
3	0	8

a

Each notation is in fact the abstract representation of a function $\sigma : \{\texttt{b}, \texttt{i}, \ldots\} \to \mathit{Val}$ given by, e.g., $\sigma(\texttt{b}) = \mathit{true}$, $\sigma(\texttt{i}) = 5$, ...

Plan

Value dimension

Data type = values (constants) + operations

Each value is represented using a memory space.

For the values of each data type, the dimension/size of representation must be mentioned.

There are two ways to define the dimension of values:

- uniform: $|v|_{\mathrm{unif}}$
- logarithmic: $|v|_{\log}$
- linear: $|v|_{\mathrm{lin}}$

Scalars

primitive types: booleans, integers, floating point numbers, strings,. . .

An important feature of these values is that they have finite representations.

Question: the irrational numbers, e.g., $\sqrt{2}$, could be scalars?

Scalars (cont)

- integers:
  $Int = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
  - uniform dimension: $|n|_{\mathrm{unif}} = 1$
  - logarithmic dimension: $|n|_{\mathrm{log}} = \log_2 abs(n)$
  - linear dimension: $|n|_{\mathrm{lin}} = abs(n)$
- booleans:
  $Bool = \{false, true\}$
  - uniform dimension: $|b|_{\mathrm{unif}} = 1$
  - logarithmic dimension: $|b|_{\mathrm{log}} = 1$
  - linear dimension: $|b|_{\mathrm{lin}} = 1$
- floating point numbers:
  $Float =$ rational numbers
  - uniform dimension: $|v|_{\mathrm{unif}} = 1$
  - logarithmic dimension: $|v|_{\mathrm{log}} = \log_2(\mathrm{mantisă}) + \log_2(\mathrm{exponent})$
  - linear dimension: $|v|_{\mathrm{lin}} = \mathrm{mantisă} \times 10^{\mathrm{exponent}} + \mathrm{exponent}$
- ...

We have $Int \cup Bool \cup Float \cup \ldots \subseteq Val$.

Arrays

- $a = [a_0, a_1, \ldots, a_{n-1}]$
- $|a|_d = |a_0|_d + |a_1|_d + \cdots + |a_{n-1}|_d$, $d \in \{\mathrm{unif}, \log, \mathrm{lin}\}$
- $Arr_n\langle V\rangle = \{\{[a_0, a_1, \ldots, a_{n-1}] \mid v_i \in V, i = 0, \ldots, n-1\}$
- $\bigcup_{n\geq 1} Arr_n\langle V\rangle \subset Val$ for each data-type $V \subset Val$
- bidimensional arrays are arrays of unidimensional arrays,
- tridimensional arrays are arrays of bidimensional arrays,
- etc.

Structures

Example: the plane point $(2, 7)$ is represented by the structure

$$\{\mathtt{x} \to 2\ \mathtt{y} \to 7\}.$$

$F = \{f_1, \ldots, f_n\}$
$s = \{f_1 \to v_1, \ldots, f_n \to v_n\}$

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

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

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

$Str\langle f_1{:}V_1, \ldots f_n{:}V_n\rangle \subset Val$ for each structure $F = \{f_1{:}V_1, \ldots f_n{:}V_n\}$.

Linear lists

A list value is a sequence $l = \langle v_0, v_1, \ldots, v_{n-1} \rangle$.

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

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

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

We have $LLin\langle V \rangle \subset Val$ for each data type V.

Complex values: graphs

The graph $G = (\{0, 1, 2, 3\}, \{(0, 1), (0, 2), (0, 3), (1, 2)\})$ is represented by the following value (using the external adjacency lists):

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

Plan

Data type (cont.)

Data type = objects + operations

Each operation *op* has a time cost $time(op)$.

For each operation of any data type must the cost time must be mentioned.

There three ways to measure the time (inherited from the value dimension):

uniform: $time_{\mathrm{unif}}(op)$ – uses the uniform dimension of values
logarithmic: $time_{\log}(op)$ – uses the logarithmic dimension of values
linear: $time_{\mathrm{lin}}(op)$ – uses the linear dimension of values

Operations with scalars

Integers:

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

The linear case: on the blackboard.

Arrays

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

where $A \mapsto [a_0, \ldots, a_{n-1}]$

The linear case: on the blackboard.

Structures

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

where $S \mapsto \{\ldots x \to s_x, \ldots\}$

Linear lists: operations definition

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

where $L \mapsto [v_0, \ldots, v_{n-1}]$

Linear lists: operations (version 1)

- corresponds to the implementations with arrays

Operation	$time_{\mathrm{unif}}(op)$	$time_{\log}(op)$
L.lookup(i)	$O(1)$	$O(\log i + \lvert v_i \rvert_{\log})$
L.insert(i, x)	$O(L.\mathtt{size}() - i)$	$O(\log i + \lvert x \rvert_{\log})$
L.remove(i)	$O(L.\mathtt{size}() - i)$	$O(\log i + \lvert v_i \rvert_{\log} + \cdots + \lvert v_{n-1} \rvert_{\log})$
...	...	...

The linear case: on the blackboard.

Linear lists: operations (version 2)

- corresponds to the implementation with double linked lists

Operation	$time_{\mathrm{unif}}(op)$	$time_{\log}(op)$
L.`lookup`(i)	$O(i)$	$O(\log(1 + \cdots + i) + \lvert v_i \rvert_{\log})$
L.`insert`(i, x)	$O(i)$	$O(\log(1 + \cdots + i) + \lvert x \rvert_{\log})$
L.`remove`(i)	$O(i)$	$O(\log(1 + \cdots + i))$
...	...	...

The linear case: on the blackboard.

Plan

Expressions: syntax

Similar to that of C++:

- arithmetic expressions: `a * b + 2`
- relational expressions: `a < 5`
- boolean expressions: `(a < 5) && (a > -1)`
- set expressions: `s1 U s2`   `s1 ^ s2`   `s1 \ s2`
- function call: `f(a*2, b+5)`
- operation call for lists/array/...: `l.update(2,55)`   `l.size()`

Instructions: syntax

- assignment: `a =` E`;`   `a[i] =` E`;`   `p.x =` E`;`

Instructions: syntax

- assignment: `a =` E`;`   `a[i] =` E`;`   `p.x =` E`;`
- function call: `quicksort(a);`   `l.insert(2,77);`

Instructions: syntax

- assignment: `a =` E`;`   `a[i] =` E`;`   `p.x =` E`;`
- function call: `quicksort(a);`   `l.insert(2,77);`
- block: `{` Sts `}`

Instructions: syntax

- assignment: `a =` E`;` `a[i] =` E`;` `p.x =` E`;`
- function call: `quicksort(a);` `l.insert(2,77);`
- block: `{` Sts `}`
- conditional instructions:
  `if (` E `)` St
  `if (` E `)` St_1 `else` St_2

Instructions: syntax

- assignment: `a =` E`;`   `a[i] =` E`;`   `p.x =` E`;`
- function call: `quicksort(a);`   `l.insert(2,77);`
- block: `{` Sts `}`
- conditional instructions:
  `if (` E `)` St
  `if (` E `)` St_1 `else` St_2
- iterative instructions:
  `while (` E `)` St

Instructions: syntax

- assignment: `a =` E`;`   `a[i] =` E`;`   `p.x =` E`;`
- function call: `quicksort(a);`   `l.insert(2,77);`
- block: `{` Sts `}`
- conditional instructions:
  `if (` E `)` St
  `if (` E `)` St_1 `else` St_2
- iterative instructions:
  `while (` E `)` St
  `forall` X `in` S St

Instructions: syntax

- assignment: `a =` E`;` `a[i] =` E`;` `p.x =` E`;`
- function call: `quicksort(a);` `l.insert(2,77);`
- block: `{` Sts `}`
- conditional instructions:
  `if (` E `)` St
  `if (` E `)` St_1 `else` St_2
- iterative instructions:
  `while (` E `)` St
  `forall` X `in` S St
  `for (` X `=` E`;` E'`; ++`X `)` S

Instructions: syntax

- assignment: `a =` E`;`   `a[i] =` E`;`   `p.x =` E`;`
- function call: `quicksort(a);`   `l.insert(2,77);`
- block: `{` Sts `}`
- conditional instructions:
  `if (` E `)` St
  `if (` E `)` St_1 `else` St_2
- iterative instructions:
  `while (` E `)` St
  `forall` X `in` S St
  `for (` X `=` E`;` E'`;` `++`X `)` S
- return: `return` E`;`

Instructions: syntax

- assignment: `a =` E`;` `a[i] =` E`;` `p.x =` E`;`
- function call: `quicksort(a);` `l.insert(2,77);`
- block: `{` Sts `}`
- conditional instructions:
  `if (` E `)` St
  `if (` E `)` St_1 `else` St_2
- iterative instructions:
  `while (` E `)` St
  `forall` X `in` S St
  `for (` X `=` E`;` E'`; ++`X `)` S
- return: `return` E`;`
- sequential composition: St_1 St_2

Alk is extendable: it can be added new data type and operations, mentioning the dimensions and resp. the time costs.

Data types

Are predefined in Alk.

It does not exists variable declarations; we assume that there is some meta-information mentioning the type of each variable.

Example of 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 verices 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);
```

Semantics: Expressions valuation

Consider a function $[\![\_]\!](\_) : \textit{Expresii} \to (\textit{Stare} \to \textit{Valori})$, where $[\![E]\!](\sigma)$ return the value of the expression E computed in the state σ.

Example: Let σ be a state that includes $\mathtt{a} \mapsto 3 \;\; \mathtt{b} \mapsto 6$. We have:
$[\![\mathtt{a} + \mathtt{b} * 2]\!](\sigma) =$

Semantics: Expressions valuation

Consider a function $[\![\_]\!](\_) : \mathit{Expresii} \to (\mathit{Stare} \to \mathit{Valori})$, where $[\![E]\!](\sigma)$ return the value of the expression E computed in the state σ.

Example: Let σ be a state that includes $\mathtt{a} \mapsto 3 \;\; \mathtt{b} \mapsto 6$. We have:
$[\![\mathtt{a} + \mathtt{b} * 2]\!](\sigma) =$
$[\![\mathtt{a}]\!](\sigma) +_{Int} [\![\mathtt{b} * 2]\!](\sigma) =$

Semantics: Expressions valuation

Consider a function $[\![\_]\!](\_) : \mathit{Expresii} \to (\mathit{Stare} \to \mathit{Valori})$, where $[\![E]\!](\sigma)$ return the value of the expression E computed in the state σ.

Example: Let σ be a state that includes $\mathtt{a} \mapsto 3$ $\mathtt{b} \mapsto 6$. We have:
$[\![\mathtt{a} + \mathtt{b} * 2]\!](\sigma) =$
$[\![\mathtt{a}]\!](\sigma) +_{\mathit{Int}} [\![\mathtt{b} * 2]\!](\sigma) =$
$3 +_{\mathit{Int}} [\![b]\!](\sigma) *_{\mathit{Int}} [\![2]\!](\sigma) =$

Semantics: Expressions valuation

Consider a function $[\![\_]\!](\_) : \textit{Expresii} \to (\textit{Stare} \to \textit{Valori})$, where $[\![E]\!](\sigma)$ return the value of the expression E computed in the state σ.

Example: Let σ be a state that includes $\mathtt{a} \mapsto 3\ \ \mathtt{b} \mapsto 6$. We have:
$[\![\mathtt{a} + \mathtt{b} * 2]\!](\sigma) =$
$[\![\mathtt{a}]\!](\sigma) +_{Int} [\![\mathtt{b} * 2]\!](\sigma) =$
$3 +_{Int} [\![b]\!](\sigma) *_{Int} [\![2]\!](\sigma) =$
$3 +_{Int} 6 *_{Int} 2 =$

Semantics: Expressions valuation

Consider a function $[\![\_]\!](\_) : \mathit{Expresii} \to (\mathit{Stare} \to \mathit{Valori})$, where $[\![E]\!](\sigma)$ return the value of the expression E computed in the state σ.

Example: Let σ be a state that includes $\mathtt{a} \mapsto 3 \;\; \mathtt{b} \mapsto 6$. We have:

$[\![\mathtt{a} + \mathtt{b} * 2]\!](\sigma) =$
$[\![\mathtt{a}]\!](\sigma) +_{\mathit{Int}} [\![\mathtt{b} * 2]\!](\sigma) =$
$3 +_{\mathit{Int}} [\![b]\!](\sigma) *_{\mathit{Int}} [\![2]\!](\sigma) =$
$3 +_{\mathit{Int}} 6 *_{\mathit{Int}} 2 =$
$3 +_{\mathit{Int}} 12 = 15$

where $+_{\mathit{Int}}$ represents the algorithm for integer addition and $*_{\mathit{Int}}$ represents the algorithm for integer multiplication.

Time cost for evaluation

$time_d([\![\texttt{a} + \texttt{b} * 2]\!](\sigma)) =$
$time_\texttt{d}([\![\texttt{a}]\!](\sigma)) + time_\texttt{d}([\![\texttt{b}]\!](\sigma)) + time_\texttt{d}(6 *_{Int} 2) + time_\texttt{d}(3 +_{Int} 122)$,
$d \in \{\mathrm{unif}, \mathrm{log}, \mathrm{lin}\}$.

$\sigma = \texttt{a} \mapsto 3 \;\; \texttt{b} \mapsto 6$

$time_{\log}([\![\texttt{a}]\!](\sigma)) = \log 3,\ time_{\log}([\![\texttt{b}]\!](\sigma)) = \log 6$
$time_{\mathrm{unif}}([\![\texttt{a}]\!](\sigma)) = 1,\ time_{\mathrm{unif}}([\![\texttt{b}]\!](\sigma)) = 1$
$time_{\mathrm{lin}}([\![\texttt{a}]\!](\sigma)) = 3,\ time_{\mathrm{lin}}([\![\texttt{b}]\!](\sigma)) = 6$

Semantics: Configurations

A configuration is a pair $\langle \textit{piece-of-program}, \textit{state} \rangle$

Example:

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

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

Semantics: Execution steps

An execution step is a transition relation between configurations:

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

iff

executing the first instruction from S in the state σ we obtain the piece of prgram S', which follows to be executed in the state σ'

Semantics: Execution steps

An execution step is a transition relation between configurations:

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

iff

executing the first instruction from S in the state σ we obtain the piece of prgram S', which follows to be executed in the state σ'

Execution steps are described by rules $\langle S_1, \sigma_1 \rangle \Rightarrow \langle S_2, \sigma_2 \rangle$, where $S_1, S_2, \sigma_1, \sigma_2$ are terms with variables (patterns).

Semantics: Execution steps

An execution step is a transition relation between configurations:

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

iff

executing the first instruction from S in the state σ we obtain the piece of prgram S', which follows to be executed in the state σ'

Execution steps are described by rules $\langle S_1, \sigma_1 \rangle \Rightarrow \langle S_2, \sigma_2 \rangle$, where $S_1, S_2, \sigma_1, \sigma_2$ are terms with variables (patterns).

To compute the time of an execution step, we describe how compute the time for each rule application.

Semantics: Assignment

assignment: `x = ` E`;`

- *informal*: evaluate E and assign the result to the variable `x`

Semantics: Assignment

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

- *informal*: evaluate E and assign the result to the variable $\mathtt{x}$
- *formal*:
  $\langle \mathtt{x}\ =\ E\,; S, \sigma\}\rangle \Rightarrow \langle S, \sigma'\rangle$
  where σ of the form $\ldots \mathtt{x} \mapsto v \ldots$ and σ' dof the form $\ldots \mathtt{x} \mapsto [\![\mathrm{E}]\!](\sigma) \ldots$ (the rest is the same as in σ).

Semantics: Assignment

assignment: `x = E;`

- *informal*: evaluate E and assign the result to the variable $\mathtt{x}$
- *formal*:
  $\langle \mathtt{x\ =\ } E\,; S, \sigma\}\rangle \Rightarrow \langle S, \sigma'\rangle$
  where σ of the form $\ldots \mathtt{x} \mapsto v \ldots$ and σ' dof the form $\ldots \mathtt{x} \mapsto [\![\mathrm{E}]\!](\sigma) \ldots$ (the rest is the same as in σ).

Time cost:
$time_d(\langle \mathtt{x\ =\ } E\,; S, \sigma\rangle \Rightarrow \langle S, \sigma'\rangle) = time_{\log}([\![E]\!](\sigma) + |[\![E]\!](\sigma))_d$
where $d \in \{\mathrm{unif}, \log, \mathrm{lin}\}$.

Semantics: `if` command

if: $\texttt{if (}E\texttt{) then } S \texttt{ else } S'$

- *informal*: evaluate e; if the result is *true*, then execute S, else execute S'

Semantics: `if` command

if: $\texttt{if (}E\texttt{) then } S \texttt{ else } S'$

- *informal*: evaluate e; if the result is *true*, then execute S, else execute S'
- *formal*:

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

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

Semantics: `if` command

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

- *informal*: evaluate e; if the result is *true*, then execute S, else execute S'
- *formal*:

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

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

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

Semantics: `while` command

while: `while (`E`)` S

- *informal*: evaluate e; if the result is *true*, then execute S, then evaluate again e and ...; otherwise the execution of the instruction stops

Semantics: `while` command

while: `while (`E`)` S

- *informal*: evaluate e; if the result is *true*, then execute S, then evaluate again e and ...; otherwise the execution of the instruction stops
- *formal*: it is described using `if`:
  $\langle \texttt{while (}e\texttt{)}\ S\ \ S', \sigma\rangle \Rightarrow$
  $\langle \texttt{if (}e\texttt{) \{ } S \texttt{ ; while (}e\texttt{) } S \texttt{ \} else \{ \}} S', \sigma\rangle$

Semantics: `while` command

while: `while (`E`)` S

- *informal*: evaluate e; if the result is *true*, then execute S, then evaluate again e and $\ldots$; otherwise the execution of the instruction stops
- *formal*: it is described using `if`:
  $\langle \texttt{while (}e\texttt{)}\ S\quad S', \sigma\rangle \Rightarrow$
  $\langle \texttt{if (}e\texttt{) \{}\ S\ \texttt{; while (}e\texttt{)}\ S\ \texttt{\} else \{ \}}S', \sigma\rangle$

Time cost:
$time_d(\langle \texttt{while (}E\texttt{) then}\ S\ \texttt{else}\ S'\ S, \sigma\rangle \Rightarrow \langle \texttt{if (}e\texttt{)}\ \ldots S, \sigma\rangle) = 0$,
$d \in \{\mathrm{unif}, \log, \mathrm{lin}\}$.

Semantics: Function call

Consider `f(a,b)` $\{\ S_f\ \}$.

We have to add stacks to the configurations.

The evaluation $\texttt{f}(e_1,e_2)$ consists of:

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

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

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

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

Assumption: the time cost of a function call is the sum of time for parameters evaluation and the time for executing the function body.

Computation (execution)

A computation (an execution) is a sequence of execution steps:
$\tau = \langle S_1, \sigma_1 \rangle \Rightarrow \langle S_2, \sigma_2 \rangle \Rightarrow \langle S_3, \sigma_3 \rangle \Rightarrow \ldots$

Computation (execution)

A computation (an execution) is a sequence of execution steps:
$\tau = \langle S_1, \sigma_1 \rangle \Rightarrow \langle S_2, \sigma_2 \rangle \Rightarrow \langle S_3, \sigma_3 \rangle \Rightarrow \ldots$

The cost of a computation:
$time_d(\tau) = \sum_i time_d(\langle S_i, \sigma_i \rangle \Rightarrow \langle S_{i+1}, \sigma_{i+1} \rangle)$, $d \in \{unif, log, lin\}$

Computation: example

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

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

The cost:
uniform cost: 3 (= the number of steps)
logarithmic cost: $\log 7 + \log 7 + \log 12 + \log 19 + \log 4$
linear cost: $7 + 7 + 12 + 19 + 4$

Computation: example

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

We used:
$[\![\texttt{i} > 5]\!](\texttt{i} \mapsto 6 \ \texttt{x} \mapsto 12) = \textit{true}$
$[\![\texttt{i} - -]\!](\texttt{i} \mapsto 6 \ \texttt{x} \mapsto 12) = 0$
$[\![\texttt{i} > 5]\!](\texttt{i} \mapsto 5 \ \texttt{x} \mapsto 12) = \texttt{false}$

The cost:
uniform cost: 5 (= numărul de pași)
logarithmic cost: $\log 6 + \log 6 + \log 5 + \log 5$

Plan

Running the algorithm DFS recursive

To execute the above algorithm on the digraph:

$$
\begin{aligned}
D.n &= 3, \\
D.a[0] &= \langle 1, 2 \rangle \\
D.a[1] &= \langle 2, 0 \rangle \\
D.a[2] &= \langle 0 \rangle
\end{aligned}
$$

create a file "dfs.in" with the following contents:

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

and then execute the algorithm with this input:

```
> alki -a dfsrec.alk  -i dfs.in
[1, 1, 1]
```

Demo