

# Alk Primer

(Draft)

Java-Semantics Version

Dorel Lucanu

February 17, 2019

# Chapter 1

## Introduction

### 1.1 Motivation

Alk is an algorithmic language intended to be used for teaching data structures and algorithms using an abstraction notation (independent of programming language).

The goal is to have a language that:

- is simple to be easily understood;
- is expressive enough to describe a large class of algorithms from various problem domains;
- is abstract: the algorithm description must make abstraction of implementation details, e.g., the low-level representation of data;
- is a good mean for learning how to algorithmically think;
- supply a rigorous computation model suitable to analyse algorithms;
- is executable: the algorithm can be executed, even if they are partially designed;
- is accompanied by a set of tools helping to analyse the algorithm correctness and the efficiency;
- input and output are given as abstract data types, ignoring implementation details.

As a starting example we consider the Alk description of the Euclid algorithm:

```
gcd(a, b)
{
    while (a != b) {
        if (a > b) a = a - b;
        if (b > a) b = b - a;
    }
    return a;
}
```

The algorithm is described using a C++-like notation. The name of the algorithm is `gcd` and its input parameters are `a` and `b`. There is no need to declare the type of parameters and/or the type of the return value. In order to execute the `gcd` algorithm, just add a single line algorithm

```
print(gcd(12, 8));
```

and execute it ("`gcd.alk`" is the file including the above code):<sup>1</sup>

---

<sup>1</sup>In this document "alki" denotes one the two scripts running the Alk interpretes: "alki.bat" (for Windows platform), respectively "alki.sh" (for linux, Mac OS).

```
> alki gcd.alk
4
```

An alternative is to write a general call of the algorithm

```
print(gcd(u, v));
```

and mention the initial values of the global variables  $u$  and  $v$  in an input file, say "gcd.in":

```
u |-> 42 v |-> 56
```

The input file is given as a parameter of the command line:

```
> alki gcd.alk gcd.in
14
```

A more complex algorithm is the DFS traversal of a digraph represented with external adjacent lists:

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

```
// the calling algorithm
dfs(D, i0) {
  i = i0;
  while (i < D.n) {
    S[i] = 0;
    i = i + 1;
  }
  dfsRec(D, i0, S);
  return S;
}
print(dfs(D, i0));
```

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 dfsrec.alk dfs.in
[1, 1, 1]
```

**Remark.** *This is a in progress document that is incrementally updated.*

## Chapter 2

# Language Description

The examples used in this manual can be found in the folder "doc/examples-from-manual".

### 2.1 Variables and their Values

Alk includes two categories of values: 3ex

*Scalars (primitive values).* Here are included the booleans, integers, rationals (floats), and strings.

*Structured values.* Here are included the sequences (linear lists), arrays, structures.

Note that a data can be as complex as possible, i.e, we may have arrays of sequences, arrays of arrays, sequences of arrays of structures, structures of arrays and lists, and so on.

#### 2.1.1 Scalars

The scalars are written using a syntax similar to that from the most popular programming languages:

```
index = 234;
isEven = true;
radius = 21.468;
name = "john";
```

The execution of the above algorithm produces an output as expected:

```
234
true
21.468
john
```

#### 2.1.2 Arrays

An array value is written as a sequence surrounded by square brackets:  $[v_0, \dots, v_{n-1}]$ , where  $v_i$  is a value, for  $i = 0, \dots, n - 1$ . Here is a very simple algorithm handling arrays:

Algorithm	Output
<pre>a = [3, 5, 6, 4]; i = 1; x = a[i]; a[i+1] = x; print(x); print(a);</pre>	<pre>&gt; alk arrays.alk 5 [3, 5, 5, 4]</pre>

The multi-dimensional arrays are represented a arrays of arrays:

```
a = [ [ 1, 2, 3 ], [ 4, 5, 6 ] ];
b = a[1];
c = a[1][2];
a[0] = b;
a[1][1] = 89;
print(a); // [[4, 5, 6], [4, 89, 6]]
w = [ [ [ 1, 2], [ 3, 4 ] ], [ [ 5, 6 ], [ 7, 8 ] ] ];
x = w[1];
y = w[1][0];
z = w[1][0][1];
w[0][1][0] = 99;
print(x); // [[5, 6], [7, 8]]
print(y); // [5, 6]
print(z); // 6
print(w); // [[[1, 2], [99, 4]], [[5, 6], [7, 8]]]
```

The output is indeed the expected one:

```
> alki arraysofarrays.alk
[[4, 5, 6], [4, 89, 6]]
[[5, 6], [7, 8]]
[5, 6]
6
[[[1, 2], [99, 4]], [[5, 6], [7, 8]]]
```

### 2.1.3 Sequences (linear lists)

A sequence value is written in a similar to an array, but using angle brackets:  $\langle v_0, \dots, v_{n-1} \rangle$ , where  $v_i$  is a value, for  $i = 0, \dots, n - 1$ . The list of operations over sequences includes:

<code>emptyList()</code>	returns the empty list $\langle \rangle$
<code>L.topFront()</code>	returns $v_0$
<code>L.topBack()</code>	returns $v_{n-1}$
<code>L.at(i)</code>	returns $v_i$
<code>L.insert(i, x)</code>	returns $\langle \dots v_{i-1}, x, v_i, \dots \rangle$
<code>L.removeAt(i)</code>	returns $\langle \dots v_{i-1}, v_{i+1}, \dots \rangle$
<code>L.removeAllEqTo(x)</code>	returns $L$ , where all elements $v_i$ equal to $x$ were removed
<code>L.size()</code>	returns $n$
<code>L.popFront()</code>	returns $\langle v_1, \dots, v_{n-1} \rangle$
<code>L.popBack()</code>	returns $\langle v_0, \dots, v_{n-2} \rangle$
<code>L.pushFront(x)</code>	returns $\langle x, v_0, \dots, v_{n-1} \rangle$
<code>L.pushBack(x)</code>	intoarce $\langle v_0, \dots, v_{n-1}, x \rangle$
<code>L.update(i, x)</code>	returns $\langle \dots v_{i-1}, x, v_{i+1}, \dots \rangle$

Example:

Algorithm	Output
<pre> l1 = &lt; 8, 3, 9, 4, 5, 4 &gt;; i = 1; x = l1.at(i + 1); y = l1.topFront(); print(x); // 9 print(y); // 8 l1.insert(2, 22); l1.update(3, 33); print(l1); // &lt; 8, 3, 22, 33, 4, 5, 4 &gt; l2 = l1; l2.removeAt(0); l2.removeAt(3); print(l2); // &lt; 3, 22, 33, 5, 4 &gt; l2.removeAllEqTo(4); print(l2); // &lt; 3, 22, 33, 5 &gt; </pre>	<pre> &gt; alki seq.alk 9 8 [8, 3, 22, 33, 4, 5, 4] [3, 22, 33, 5, 4] [3, 22, 33, 5] </pre>

Now we may define sequences of arrays:

Algorithm	Output
<pre> l = &lt; [1, 2, 3], [4, 5] &gt;; a = l.at(1); l.pushBack(a); </pre>	<pre> a  -&gt; [ 4, 5 ] l  -&gt; &lt; ([ 1, 2, 3 ]), [ 4, 5 ], [ 4, 5 ] &gt; </pre>

and arrays of structures:

Algorithm	Output
<pre> a = [ &lt; 1, 2, 3 &gt;, &lt; 4, 5, 6 &gt;, &lt; 7, 8 &gt; ]; b = a[1]; a[1].update(1, 11); a[0].insert(2,22); a[2] = a[1].pushBack(77); </pre>	<pre> a  -&gt; [ (&lt; 1, 2, 22, 3 &gt;), (&lt; 4, 11, 6 &gt;), (&lt; 4, 11, 6, 77 &gt;) ] b  -&gt; &lt; 4, 5, 6 &gt; </pre>

## 2.1.4 Structures

A structure value is of the form  $\{f_1 \rightarrow v_1 \dots f_n \rightarrow v_n\}$ , where  $f_i$  is a field name and  $v_i$  is a value, for  $i = 1, \dots, n$ .

Example:

Algorithm	Output
<pre> s = { x -&gt; 12  y -&gt; 45 }; a = s.x; s.y = 99; b.x = 22; </pre>	<pre> a  -&gt; 12 b  -&gt; { x -&gt; 22 } s  -&gt; { (x -&gt; 12) (y -&gt; 99) } </pre>

Note that the structure **b** has been created with only one field, because there is no information about its type, which is deduced on the fly during the execution.

We may have structures of arrays

### Algorithm

```
s = { x -> [ 1, 2, 3 ] y -> [ 4, 5, 6 ] };
b = s.y;
(s.x)[1] = 11;
```

### Output

```
b |-> [ 4, 5, 6 ]
s |-> { (x -> ([ 1, 11, 3 ])) (y -> [ 4, 5, 6 ]) }
```

sequences of arrays

### Algorithm

```
l = < { x -> 12 y -> 56 }, { x -> -43 y -> 98 }, { x -> 33 y -> 66 } >;
u = l.topFront();
l.pushBack({ x -> -100 y -> 200 });
l.at(2).x = 999;
```

### Output

```
a |-> [ 4, 5 ]
l |-> < ([ 1, 2, 3 ]), [ 4, 5 ], [ 4, 5 ] >
```

and so on.

**TODO.** Change the notation  $x \rightarrow v$  into  $x : v$ .

## 2.1.5 Sets

A set value is written as  $\{v_0, \dots, v_{n-1}\}$ , where  $v_i$  is a value, for  $i = 0, \dots, n - 1$ . The operations over sets include the union  $\cup$ , the intersection  $\cap$ , the difference  $\setminus$ , and the membership test  $\_ \text{ belongsTo } \_$ . Example:

### Algorithm

```
s1 = { 1 .. 5 };
s2 = { 2, 4, 6, 7 };
a = s1  $\cup$  s2 ;
b = s1  $\cap$  s2;
c = s1  $\setminus$  s2;
t = 2 belongsTo b  $\cap$  c;
```

### Output

```
a |-> { 7, 6, 1, 2, 3, 4, 5 }
b |-> { 2, 4 }
c |-> { 1, 3, 5 }
s1 |-> { 1, 2, 3, 4, 5 }
s2 |-> { 2, 4, 6, 7 }
t |-> false
```

Obviously, we may have sets of arrays, sequences of sets, and so on.

**Remark.** The current implementation does check if a set value assigned to a variable is indeed a set. But the operations returns sets whenever the arguments are sets.

## 2.1.6 Specification of values

Alk includes several sugar syntax mechanisms for specifying values in a more compact way:

### Algorithm

```
p = 3;
q = 9;
a = [ i | i : p .. q ];
p = 2;
b = [ a[i] | i : p .. p+3 ];
l = < b[i] * 2 | i : p-2 .. p >;
```

### Output

```
a |-> [ 3, 4, 5, 6, 7, 8, 9 ]
b |-> [ 5, 6, 7, 8 ]
l |-> < 10, 12, 14 >
p |-> 2
q |-> 9
```

### 2.1.7 Flexibility in Using Data Structures

Even if each data structure has its own operations with specific notations, these operations and notations can be "exported" to the other data structures as sugar syntax. For instance, if  $L$  is a sequence, then we may use  $L[i]$  for  $L.at(i)$ .

Example:

Algorithm	Output
<pre>L = &lt; 8, 3, 9, 4 &gt;; x = 11[2]; L[3] = 33; L[4] = 44;</pre>	<pre>L  -&gt; &lt; 8, 3, 9, 33, 44 &gt; x  -&gt; 9</pre>

If  $S$  is a set, then we can borrow the notations from sequences and array operations:

$S[i]$  – returns an element from  $S$ . The same do  $S.topFront()$  and  $S.topBack()$ . Note that the behaviour of these operations could be nondeterministic: for the same set value, they could return different elements in different execution steps;

$S.insert(i, x)$  – adds  $x$  to  $S$  (it is a short notation for  $S = S \cup \{x\}$ );

Example:

Algorithm	Output
<pre>s = { 2, 4, 6, 7 }; z = s.topFront(); n = s.size(); s2 = s.pushBack(99); a = s2[3]; s.insert(2, 22);</pre>	<pre>a  -&gt; 7 n  -&gt; 4 s2  -&gt; { 2, 4, 6, 7, 99 } s  -&gt; { 2, 4, 22, 6, 7 } z  -&gt; 2</pre>

Here is an example that shows that  $s.topFront()$  returns different values, even if it is called for equal sets:

Algorithm	Output
<pre>s = { 2, 4, 6, 7 }; s2 = s; a = s.topFront(); s2.popFront(); s2.pushBack(a); b = s2.topFront(); t = s == s2;</pre>	<pre>a  -&gt; 2 b  -&gt; 4 s2  -&gt; { 4, 6, 7, 2 } s  -&gt; { 2, 4, 6, 7 } t  -&gt; true</pre>

We can see that  $s$  and  $s2$  are equal as sets, but  $a$  and  $b$  are different.

### 2.1.8 Lists with iterators

An iterator  $p$  associated with a list  $L$  if  $p$  "refers" an element of  $L$ . With iterators one can call operation over the associated lists and/or traverse the associated list.

Operations with iterators:

$p + i$  – returns an iterator referring the  $i$ th element after  $p$ ;

$p - i$  – returns an iterator referring the  $i$ th element in front of  $p$ ;

$++p$  – moves  $p$  to the previous element (if any);

$--p$  – moves  $p$  to the next element (if any);

$L.first()$  – returns an iterator that refers the first element of  $L$ ;



`L.end()` – returns an invalid iterator for  $L$

Using iterators, one can access the elements of the lists and/or execute operations on the associated list:

`*p` – returns the referred element;

`p->delete()` – remove the element referred by  $p$  and move  $p$  to the next element;

`p->insert(x)` – insert  $x$  immediately after the element referred by  $p$ .

If  $p$  refers the  $i$ th element in  $L$ , then `p->delete()` is equivalent to `L[i].delete()` and `p->insert(x)` with `L[i].insert(x)`.

The following operators are useful to traverse circular lists:

`p +% i` – returns an iterator referring the  $i$ th element after  $p$  modulo the length of the list;

`p -% i` – returns an iterator referring the  $i$ th element in front of  $p$  modulo the length of the list;

`++%p` – moves  $p$  to the previous element modulo the length of the list;

`--%p` – moves  $p$  to the next element modulo the length of the list

Example 1:

Algorithm	Output
<pre> L = &lt; 2, 5, 8 &gt;; i = 0; A = []; for (p = L.first(); p != L.end(); ++p) {     A[i] = *p ;     ++i; } </pre>	<pre> A  -&gt; [ 2, 5, 8 ] L  -&gt; &lt; 2, 5, 8 &gt; i  -&gt; 3 p  -&gt; it ( L , 3 ) </pre>

Example 2:

Algorithm	Output
<pre> L = &lt; 2, 5, 8, 33 &gt;; p = L.first(); p = p + 3; a = *p; ++% p; b = *p; q = L.first(); c = *(-% q); </pre>	<pre> L  -&gt; &lt; 2, 5, 8, 33 &gt; a  -&gt; 33 b  -&gt; 2 c  -&gt; 33 p  -&gt; it ( L , 0 ) q  -&gt; it ( L , 3 ) </pre>

## 2.2 Expressions

Alk includes the basic operators over scalars with a C++-like syntax.

Since Alk is designed with K Framework, it can be easily extended with new operators.

## 2.3 Statements

The syntax for the statements is similar to that of imperative C++.

We already have seen examples of the assignment statement. The other statements include:

### 2.3.1 Block

Syntax: `{ Stmt }`

### 2.3.2 if

Syntax: 1) `if ( Exp ) Stmt else Stmt` 2) `if ( Exp ) Stmt`

### 2.3.3 while

Syntax: `while ( Exp ) Stmt`

### 2.3.4 for

Syntax: 1) `forall Id in Exp Stmt` 2) `for ( VarAssign ; Exp ; VarUpdate ) Stmt`

Examples:

```
for (i= 2; i <= x / 2; ++i)
  if (x % i == 0) return false;
```

```
forall y in { 1 .. 6 }
  if (y belongsTo s2) d = d U singletonSet(y);
```

### 2.3.5 Sequential Composition

Syntax: `Stmt Stmt`

## 2.4 Statements for Nondeterministic Algorithms

### 2.4.1 choose

Syntax: `choose Id in Exp ;` 2) `choose Id in Exp s.t. Exp ;`

Example 1:

Algorithm	Output
<code>choose x1 in { 1 .. 5 };</code>	<code>x1  -&gt; 3</code>
<code>choose x2 in { 1 .. 5 };</code>	<code>x2  -&gt; 1</code>

Example 2:

Algorithm	Output
<code>odd(x) {</code>	
<code>return x % 2 == 1;</code>	
<code>}</code>	<code>x1  -&gt; 5</code>
	<code>x2  -&gt; 3</code>
<code>choose x1 in { 1 .. 5 } s.t. odd(x1);</code>	
<code>choose x2 in { 1 .. 5 } s.t. odd(x2);</code>	

### 2.4.2 success

Syntax: `success ;`

Example:

Algorithm	Output
<code>odd(x) {</code>	<code>&lt;k&gt;</code>
<code>return x % 2 == 1;</code>	<code>success ;</code>
<code>}</code>	<code>&lt;/k&gt;</code>
	<code>&lt;state&gt;</code>
<code>choose a in { 1 .. 8 };</code>	<code>a  -&gt; 7</code>
<code>if (odd(a)) success;</code>	<code>&lt;/state&gt;</code>

### 2.4.3 failure

Syntax: `failure ;`

Example:

Algorithm	Output
<pre> odd(x) {   return x % 2 == 1; }  s = emptySet; for (i = 0; i &lt; 8; i = i+2)   s.pushBack(i); choose x in s s.t. odd(x); </pre>	<pre> &lt;k&gt;   failure ; &lt;/k&gt; &lt;state&gt;   i  -&gt; 8   s  -&gt; { 0, 2, 4, 6 }   x  -&gt; 6 &lt;/state&gt; </pre>

## 2.5 Functions/Procedures Describing Algorithms

Example:

Algorithm	Output for <code>n  -&gt; 5</code> <code>b  -&gt; [5,1,3,2,4]</code>
<pre> partition(out a, p, q) {   x = a[p] ;   i = p + 1;  j = q;   while (i &lt;= j) {     if (a[i] &lt;= x) i = i+1;     else if (a[j] &gt;= x) j = j-1;     else if (a[i] &gt; x &amp;&amp; x &gt; a[j]) {       swap(a, i, j);       i = i+1;       j = j-1;     }   }   k = i-1;  a[p] = a[k];  a[k] = x;   return k; }  qsort(out a, p, q) {   if (p &lt; q) {     k = partition(a, p, q);     qsort(a, p, k-1);     qsort(a, k+1, q);   } }  qsort(b, 0, n-1); </pre>	<pre> b  -&gt; [ 1, 2, 3, 4, 5 ] n  -&gt; 5 </pre>

Note that the output parameters and the input/output parameters are declared with the prefix `out`.