Langage **É**lémentaire **A**lgorithmique
Spécifications du langage

Lionel Clément

16 mars 2021

Le but de ce projet est de livrer un compilateur pour un Langage Élémentaire Algorithmique (**Léa**). Ce langage qui s'inspire de Java ou de C# est destiné à enseigner la programmation orientée objet.

Remarque : Si vous avez eu accès aux versions précédentes de ce document, il ne vous a pas échappé que **Léa** était présenté fort différemment. De fait, **Léa** n'est qu'un nom générique donné à un ensemble de langages destinés à faire des exercices de compilation.

Pour en donner une idée, voici deux petits exemples.

Exemple 1 : petit programme qui affiche les nombre de Fibonacci

fib.lea

```
/* ********************************************
* Un petit exemple de code Léa
******************************************** */
class main {
        m: map<integer, integer>;

        // Définition de la fonction principale
        main(args: list<string>) {
                foreach i in [1 .. 20]
                        writeln (fib(i));
                return 0;
        }

        private function fib(n: integer): integer {
                if ((n == 1) || (n == 2))
                        return 1;
                else {
                        if (m.get(n) == none)
                                m.put(n, fib(n-1) + fib(n-2));
                        return m.get(n);
                }
        }
}
```

Exemple 2 : petit programme de manipulation d'objets

deQueue.lea

```
interface DeQueue<T> {
   procedure pushFront(data: integer);
   function top(): integer;

}
```

node.lea

```
class Node<T> {
   data: T;
   next: Node<T>;

   Node(data: T, next: Node<T>) {
      this.data := data;
      this.next := next;
   }
}
```

linkedListDeQueue.lea

```
import "deQueue.lea";
class LinkedListDeQueue<T> implements DeQueue<T> {
   first, last: Node<T>;

   LinkedListDeQueue () {
      first := last := null;
   }

   procedure pushFront(data: T){
      first := new Node(data, first);
      if (last == null)
         last := first;
   }

   function popFront(): T{
      result: T := first;
      first := first.next;
      return result;
   }

   function top(): integer{
      return first.data;
   }

}
```

stack.lea

```
import "linkedListDeQueue.lea";
class Stack<T> extends LinkedListDeQueue<T> {

   procedure push(data: T){
      pushFront(data);
   }

   function pop(): T{
      return popFront();
   }

}
```

main.lea

```
import "stack.lea";
class main {
   // Définition de la fonction principale
   main(list<string> args) {

      stack: Stack<integer> := new Stack();

      stack.push(3);
      stack.push(5);

      writeln (stack.top());

      return 0;
   }
}
```

Le langage est procédural orienté objet à définition de classes. Il utilise un typage statique explicite fort. C'est-à-dire que toutes les variables sont déclarées avec un type unique connu à la compilation.

Structure générale d'un programme Léa

1. Déclaration des modules requis
2. Déclaration des interfaces et des classes

L'objet principal de l'application sera la création implicite d'une instance de la classe `main`. C'est-à-dire que la méthode principale d'une application réalisée avec `Léa` est l'exécution du constructeur de `main`.

Commentaires

Les commentaires sont comme en Java et en C++.
— sur une ligne ou en fin de ligne
`//`
— sur plusieurs lignes
`/* */`
— commentaires de documentation
`/** **/`

Identificateurs

Un identificateur est une séquence de lettres dont le signe `_` et de chiffres encodés en UTF-8. Il doit commencer par une lettre. On différencie les capitales des bas-de-casse (*minuscules*) et les accents sont acceptés comme en `C#`.

Déclarations et définitions

— Définition des interfaces :

`interface` <*ClassName*> <*InterfaceExtension*> { <*MethodSignatures*> }

exemple :

```
interface SortedSet<E extends Comparable> extends Collection<E>, Iterable<E>, Set<E> {
   function first(): E;
   function last(): E;
}
```

— Définition des classes :

`class` <*ClassName*> <*ClassExtension*> <*ClassImplementation*> {
<*Attributes*, *Methods*> }

exemple :

```
class TreeSortedSet<E> implements SortedSet<E> extends TreeSet<E> {
   function first(): E {return tree.first(); }
   virtual function foo(): E;
}
```

— Déclaration des constantes de classe :
Toute classe peut contenir des constantes. Leur type est connu par le type de l'expression.

`const` *identificateur* `:=` *expression*`;`

— Déclaration des noms de types :

`type` *identificateur* `:=` *expressiondetype*`;`

— Déclaration des attributs :

identificateur`:` *type*`;`
identificateur`:` *type* `:=` *valeur*`;`

— Déclaration des variables de classes :

`static` *identificateur*`:` *type*`;`
`static` *identificateur*`:` *type* `:=` *valeur*`;`

— Déclaration des méthodes :

`procedure` *identificateur* `(`*liste d'arguments*`);`
`function` *identificateur* `(`*liste d'arguments*`):` *type*`;`

Par défaut, les arguments de type élémentaire sont passés par valeur, les autres par référence. Pour passer un argument élémentaire par référence, on le fait précéder de `&`.

— Déclaration des procédures et fonctions de classes :

`static procedure` *identificateur* `(`*liste d'arguments*`);`
`static function` *identificateur* `(`*liste d'arguments*`):` *type*`;`

— Définition des constructeurs :

`<`*ClassName*`>` `(`*liste d'arguments*`)` *bloc*

Types

Les types suivants sont disponibles

— Types élémentaires : **char**, **integer**, **float**, **boolean**, **string**

— Type énuméré : `enum`$< T_1, \ldots, T_k >$ où T_i est un type nommé (c'est-à-dire un identificateur qui vaut pour un type)
Chaque type nommé est accessible publiquement comme une constante l'est.
Exemple :

```
class A {
   type all_colors = enum<WHITE, BLACK>;
   color: all_colors
   function getColor(): all_colors {return color;}
}

class B {
   a: A;
   function isWhite() {
      return (a.getColor() = A.WHITE);
   }
}
```

— Classes et interfaces : `class/interface` *ClassName* $[< T_1, \ldots, T_k >]$ où T_i est une variable de type (c'est-à-dire une variable dont la valeur est un type)

— Listes : `list`$< T >$ où T est un type. Une variable recevant le type `list`$< T >$ est une nouvelle instance d'un objet dont les méthodes disponibles sont les suivantes :
  — procedure add(t : T)
  — function get(i : integer)

  - procedure clear()
  - function contains(t : T) : boolean
  - function isEmpty() : boolean
  - function iterator() : iterator<T>
  - function size() : integer
- Intervalle : `range`$< T >$. Une variable recevant le type `range`$< T >$ est une nouvelle instance d'un objet dont les méthodes disponibles sont les suivantes :
  - function first() : T
  - function last() : T
  - function contains(t : T) : boolean
  - function size() : integer
  - function hasNext() : boolean
  - function next() : T

  T est une classe qui implémente *comparable* $< U >$ ou un type de base.
  Exemple :

```
class A {
   r: range<integer> := [12 .. 36];
   function foo(): {
       writeln(r.first()); // 12
       writeln(r.last()); // 36
       writeln(r.contains(15)); // true
       writeln(r.size()); // 25
       while (r.hasNext()) {
         writeln(r.next()); // 12 13 14 ... 36
      }
   }
}
```

- Ensembles : `set`$< T >$. Une variable recevant le type `range`$< T >$ est une nouvelle instance d'un objet dont les méthodes disponibles sont les suivantes :
  - procedure add(t : T)
  - function remove(t : T) : boolean
  - procedure clear()
  - function contains(t : T) : boolean
  - function isEmpty() : boolean
  - function iterator() : iterator<T>
  - function size() : integer

  T est une classe qui implémente *equivalent* $< U >$ ou un type de base.
- Applications : `map`$< K, V >$. Une variable recevant le type `range`$< T >$ est une nouvelle instance d'un objet dont les méthodes disponibles sont les suivantes :
  - function put(k : K, v : V) : boolean
  - function get(k : K) : V
  - function remove(k : K) : boolean
  - procedure clear()
  - function containsKey(k : K) : boolean
  - function containsValue(v : V) : boolean
  - function isEmpty() : boolean
  - function iterator() : iterator<K>
  - function size() : integer

  K est une classe qui implémente *equivalent* $< T >$ ou un type de base .

Interfaces prédéfinies

- Éléments comparables : `comparable`$< T >$ où T est un type. Cette interface permet de définir un ordre sur les objets.
  La méthode disponible est la suivante :
  - function compareTo(t : T) : integer

— Éléments équivalents : `equivalent`$< T >$ où T est un type. Cette interface permet de définir une relation d'équivalence sur les objets.
La méthode disponible est la suivante :
— function equivalentTo(t : T) : boolean
— Ensembles itérables : `iterable`$< T >$ où T est un type. Cette interface permet de définir une classe contenant un itérateur.
La méthode disponible est la suivante :
— function iterator() : Iterator<T>
— Élément itérables : `iterator`$< T >$ où T est un type. Cette interface permet de définir une classe contenant un itérateur sur un élément.
Les méthodes disponibles sont les suivantes :
— function hasNext() : boolean
— function next() : T

private, protected et public (non utilisé pour le mini-projet)

Les attributs et les méthodes d'un objet de classe `T` sont accessibles depuis un autre objet selon les règles du langage C++. La seule différence est que les valeurs par défaut des attributs est `private`, des méthodes est `public` et de l'héritage est `public`.

— Depuis un objet membre de `T` (*private*)
— Depuis un objet membre d'une classe qui hérite de `T` (*protected*)
— Depuis tout objet (*public*)

Les attributs et les méthodes d'un objet de classe `T` qui hérite d'une classe `U` sont accessibles depuis un autre objet selon ces règles :

— *private* si l'héritage est *private*
— *protected* si l'héritage est *protected*
— Inchangé si l'héritage est *public*

Exemple :

```
class A {
   public x: integer;
   protected y: integer;
   private z: integer;
   k: integer;
   procedure foo(){...}
}

class B extends public A
{
   // x is public
   // y is protected
   // z is not accessible from B
   // k is not accessible from B
   // foo is public
}

class C extends protected A
{
   // x is protected
   // y is protected
   // z is not accessible from C
   // k is not accessible from B
   // foo is public
}

class D extends private A
{
    // x is private
   // y is private
   // z is not accessible from D
   // k is not accessible from B
   // foo is public
```

```
}
```

Méthodes virtuelles virtual (non utilisé pour le mini-projet)

Une méthode déclarée mais non définie dans une classe est marquée par le mot clef **virtual**. Une classe qui contient une telle méthode est abstraite et ne peut pas être instanciée. La méthode devra être définie dans une classe qui hérite de celle-ci.

Polymorphisme (non utilisé pour le mini-projet)

Toute méthode peut être déclarée plus d'une fois dans la même interface ou la même classe avec des arguments différents. La signature de la méthode suffit à en connaître la définition.

Toute méthode peut être redéfinie dans une classe qui hérite de la classe où elle est définie une première fois.

Méthodes finales final (non utilisé pour le mini-projet)

Une méthode déclarée **final** ne peut pas être redéfinie dans les sous-classes.

Redéfinition des opérateurs (non utilisé pour le mini-projet)

Les opérateurs $++ + --- */\%+ = - = * = / =<<=>>===! = \&\&||\&|! <<>>$ [] peuvent être redéfinis comme en C++.

Exemple

```
class User {
   firstName: string;
   lastName: string;

        operator < (User first, User second) {
           if (first.firstName < second.firstName)
              return true;
           if (first.lastName < second.lastName)
              return true;
           return false;
        }
```

This

Comme en C++ ou Java, le mots clef **this** est une expression qui désigne l'instance de la classe.

Instructions et structures de contrôle

Affectation

Pour dissuader l'apprenant de confondre l'égalité avec l'affectation, nous utilisons le signe `:=`

AffectableExpr `:=` *expr*`;`

On pourra associer l'affectation d'initialisation et la déclaration des attributs :

variable`:` *type* `:=` *expr*`;`

Les affectations spéciales `+=`, `-=`, `*=`, `/=`, $\neg$`=`, $\wedge$`=`, $\vee$`=` ont les significations habituelles, où

variable `op=` *expr*`;`

vaut pour

variable `:=` *variable* `op` *expr*`;`

Appel de procédure

Si `foo(x: integer)` est une procédure, `foo(36);` est accessible comme instruction.

Remarque. Contrairement à de nombreux langages, une instruction ne peut pas être utilisée comme expression. Il n'est donc pas possible d'écrire une procédure ou une affectation comme expression.

ProcedureName `(`$expr_1, expr_2, \ldots, expr_k$`)` `;`

Entrées sorties

Les procédures `write(`*expr*`)`, `writeln(`*expr*`)` permettent un accès aux sorties standard. Le type de l'argument `expr` est un type élémentaire. L'expression `readln()` correspond à l'entrée standard (une chaîne de caractères terminée par `return`) qui renvoie un `string`.

Conditionnelles et choix multiples

Si (I, $I_1, I_2, \ldots I_k$ sont des instructions), les instructions conditionnelles suivantes sont admises :

`if (`*expr*`)` I
`if (`*expr*`)` I_1 `else` I_2
`if (`*expr*`)` I_1 `elif (`*expr*`)` I_2 `elif (`*expr*`) ...` `else` I_k

expr doit être du type `boolean` à l'exclusion de tout autre type.

`case` *expr* `{`
$enum_{slot_1} : I_1$
$enum_{slot_2} : I_2$
...
$enum_{slot_k} : I_k$
`}`

expr doit être de type `enum`, $enum_{slot_i}$ doit être un énuméré de *expr*.

Boucles

Si I est une instruction, les boucles suivantes sont admises :

1. boucle **while** et **repeat**

   `while (`*expr*`)` I
   `repeat` I `while (`*expr*`)` `;`

   Comme pour les conditionnelles, *expr* doit être du type `boolean`.

2. boucle **foreach**

   `foreach` *var* `in` *expr* I

   Expr est une expression de type
   — Liste
   — Ensemble
   — Application
   — Chaîne de caractères
   — Énuméré

L'instruction `for` déclare implicitement la variable *var* localement et son type se rapporte au type de *expr* (`char` si *expr* est un `string`, l'un des énumérés pour `enum`, élément pour la liste ou l'ensemble, etc.).

```
procedure foo(){
   k: map<integer, string>;
   s: string;
   t: enum (ROUGE, BLEU, VERT);
   for i in [1 .. 100] writeln (i); // range
   for i in [1, 2, 3] writeln (i);  // list
   for i in {1, 2, 3} writeln (i);  // set
   for i in k {
        write (j.first());
        writeln (" => " + j.second());
   }
   for i in s
        writeln(i);
   for i in t
        writeln(i);
}
```

3. **break**, **continue**

   Les boucles peuvent être interrompues par l'instruction élémentaire **break**

```
for n in [2..10]{
     for x in [2..n]{
         if (n % x == 0){
             writeln("composite number")
             break
             }
     }
     else
         writeln("prime number")
}
```

   L'instruction d'une boucle peut être interrompue par **continue**

```
for num in range(2, 10){
   if (num % 2 == 0){
        writeln ("Found an even number"+num.toString());
       continue; }
    writeln("Found a number"+num.toString());
}
```

Blocs

Un bloc est un ensemble d'instructions et de déclarations de variables locales à ces instructions. Il se note entre deux accolades.

Expressions

Les expressions utilisent les variables, les constantes élémentaires et complexes, les opérateurs et les fonctions.

On peut trouver des expressions qui ont des effets comme $x++$ ou $foo(y)$ où y est un argument passé en paramètre.

Les opérateurs et fonctions membres seront repris des langages Java ou C++ comme traditionnellement.

Nous dressons dans le tableau suivant les expressions du langage **Léa**

Expression	**Type**	**Valeur**
""	string	chaîne de caractères où sont échappés les caractères spéciaux `\t,\n,\r,\\,\"`
”	char	caractère où est échappé le caractère spécial `\’`
nombre entier	integer	nombre entier décimal signé de Integer.MIN_VALUE à Integer.MAX_VALUE
nombre flottant	float	nombre flottant décimal signé de Float.MIN_VALUE à Float.MAX_VALUE
`true`, `false`	boolean	
`none`		Objet non alloué
$E + E$	type des opérandes	addition, concaténation, union
$E + +$	type de l'opérande	incrémentation, la valeur retournée est $E + 1$
$+ + E$	type de l'opérande	incrémentation, la valeur retournée est E
$E - E$	type des opérandes	soustraction, différence
$E - -$	type de l'opérande	décrémentation, la valeur retournée est $E - 1$
$- - E$	type de l'opérande	décrémentation, la valeur retournée est E
$E * E$	type des opérandes	multiplication, intersection*
E/E	type des opérandes	division
$E\%E$	type des opérandes	modulo
$-E$	type de l'opérande	moins unaire, complément
$E < E$	boolean	inférieur
$E > E$	boolean	supérieur
$E \leq E$	boolean	inférieur ou égal
$E \geq E$	boolean	supérieur ou égal
$E == E$	boolean	égal
$E \mathrel{!}= E$	boolean	différent
$E \wedge E$	boolean	et
$E \vee E$	boolean	ou
$\neg E$	boolean	non
$< E_1, E_2, \ldots E_k >$	tuple $< type(E_1), type(E_2), \ldots, type(E_k) >$	construction d'un tuple
$E_1 \Rightarrow E_2$	tuple $< type(E_1), type(E_2) >$	construction d'une paire
$[E_1, E_2, \ldots E_k]$	list$< type(E_i) >$	construction d'une liste
$[E_1 \cdot\cdot E_k]$	range$< type(E_I) >$	construction d'une liste $[E_1, \ldots, E_k]$ où $i < j \Rightarrow E_i < E_j$ L'opérateur $<$ est donné pour les types élémentaires, il doit être défini pour les classes
$\{E_1, E_2, \ldots E_k\}$	set$< type(E_i) >$	construction d'un ensemble. Les opérateurs $<$ et $=$ sont donnés pour les types élémentaires, ils doivent être définis pour les classes
$\{key_1 => val_1, \ldots key_k => val_k\}$	map$< type(key_i), type(val_i) >$	construction d'une application. Les opérateurs $<$ et $=$ sont donnés pour les types de key_i élémentaires, ils doivent être définis pour les classes

$foo(E_1, E_2, \ldots E_k)$	$'y$ où $type(foo) =' x \rightarrow' y$	application de foo où $'x$ s'unifie avec $type(E_1) \times type(E_2) \ldots type(E_k)$
	ou objet instance de la classe foo	si foo est une classe déclarée et $foo(a_1, a_2, \ldots, a_k)$ un constructeur pour cette classe
$t[E]$	$'x$ où $type(t) = liste\ of\ 'x$	E^e élément de la liste t (E doit être de type $integer$)
$t[E]$	$'y$ où $type(t) = map\ of\ ('x,'y)$	deuxième élément du E^e élément de la relation t (E doit être de type $('x,'y)$)
$E.F$	type de F	F est une variable membre de l'objet ou de la classe E
$E.foo(E_1, E_2, \ldots E_k)$	$'y$ où $type(foo) =' x \rightarrow' y$	appel d'une fonction membre de l'objet ou de la classe E ou du type de E