# Langages et Automates — Projet
# Lea : Implémenter Flex en Bison

Devoir à rendre sur Madoc avant le 18/12/2020

## 1 Spécification du programme Lea

Le but de ce projet est d'implémenter Lea (pour ≪ Langage et automates ≫), un concurrent de Flex permettant d'interpréter des automates et des expressions rationnelles. Plus précisément, le programme Lea doit être en mesure de lire des fichiers `.lea` contenant la description de langages de type 3, et de générer le code `c` d'un programme qui demande des mots à l'utilisateur et affiche quels langages reconnaissent les mots entrés. Par exemple, la suite de commandes suivante crée un fichier `langages.lea` décrivant deux langages L1 et L2, génère et compile un programme `langages.exe` les reconnaissant en utilisant lea et gcc, puis lance le programme. L'utilisateur rentre `aabb` qui est reconnu par L1 mais pas par L2.

```
$ echo "L1 = a*b*; L2 = (ab)*;" > langages.lea
$ ./lea langages.lea -o langages.c
$ gcc langages.c -o langages.exe
$ ./langages.exe
> aabb
  reconnu par L1
  non reconnu par L2
```

## 2 Prise en main de l'existant

Pour vous simplifier la tâche, un squelette du code vous est déjà fourni. Dans sa forme actuelle, il ne permet de reconnaître que les automates déterministes, définis par une liste d'états initiaux, d'états finaux et de transitions. Des exemples de langages déjà reconnus, ainsi que de langages à reconnaître en commentaires, est fournie dans le fichier `langages.lea`.

Vous pouvez tester le programme fourni en le compilant avec la commande `make`, qui crée le programme `lea`. Vous pouvez également utiliser la commande `make langages.exe` pour créer le programme exécutable reconnaissant les langages décrits dans `langages.lea`, la commande `make all` pour compiler le programme `lea` ainsi que tous les fichiers `.lea` du répertoire, et la commande `make clean` (respectivement `make cleanall`) pour supprimer les fichiers générés, à part (respectivement y compris) les fichiers exécutables.

### 2.1 Structures de données.

Trois structures de données ont déjà été programmées pour vous aider dans votre tâche. Leur documentation complète est fournie en annexe de ce sujet.

**Ensembles** La classe `set<T>`, dans le fichier `set.hpp`, encode les ensembles finis, manipulables de manière algébrique.

La classe fournit trois opérateurs pour effectuer des calculs avec les ensembles : l'union (opérateur |), l'intersection (opérateur &), et la différence ensembliste (opérateur $-$). Il est possible d'utiliser chaque opérateur soit entre deux ensembles (par exemple $s_1|s_2$ représente l'union de $s_1$ et $s_2$) soit entre un ensemble et un élément (par exemple $s_1|x$ représente l'union de $s_1$ et $\{x\}$), et soit comme une loi de composition interne, soit en modification sur place (par exemple $s_1| = s_2$ signifie $s_1 = s_1|s_2$).

La comparaison entre les ensembles se base sur l'inclusion : $s_1 <= s_2$ signifie que $s_1$ est inclus dans, ou égal à $s_2$, $s_1 < s_2$ signifie que $s_1$ est strictement inclus dans $s_2$, $s_1 == s_2$ signifie que $s_1$ et $s_2$ contiennent les mêmes éléments, etc.

Il est possible d'accéder aux éléments d'un ensemble $s$ grâce à $s[i]$, pour $0 \leq i < s.\mathrm{size}()$, ce qui permet de faire des boucles. La notation `for(type x : s)` est également disponible. Enfin, $s.\mathrm{contains}(x)$ permet de savoir si $x \in s$.

**Automates** La structure `automaton`, définie dans le fichier `automaton.hpp` et implémentée dans le fichier `automaton.cpp`, décrit des automates finis non déterministes. Les états de l'automate sont encodés par des entiers, et l'alphabet de l'automate est composé de lettres minuscules de l'alphabet latin.

Un objet `automaton` expose un nom `name`, un ensemble d'états initiaux `initials`, un ensemble d'états accepteurs `finals`, et un ensemble de transitions `transitions` définis ci-dessous.

La structure `automaton` permet également d'obtenir l'ensemble des états (`a.get_states()`) et terminaux (`a.get_alphabet()`) entrant dans la définition de l'automate, de déterminer si un automate est déterministe (`a.is_deterministic()`), et de déterminer l'ensemble des états accessibles dans l'automate à partir des états d'un ensemble `from` en suivant des $\varepsilon$-transitions (`epsilon_accessible(from)`) ou une unique transition étiquetée `c` (`accessible(from, c)`).

**Transitions** La structure `transition`, dans le fichier `automaton.hpp`, encode les transitions dans les automates finis. Un objet `transition` expose un état de départ `start`, un terminal qui étiquette la transition `terminal` et un état d'arrivée `end`. Une $\varepsilon$-transition est caractérisée par un terminal nul, et peut être détecté en utilisant la fonction `transition::is_epsilon()`.

### 2.2 Flot d'exécution.

La fonction principale du programme est donnée dans le fichier `lea.cpp`. Dans un premier temps, les arguments de la ligne de commande sont lus, et tous les fichiers d'entrée est interprété par la fonction `read_lea_file`, qui retourne un ensemble d'automates. Ensuite, des vérifications sont faites sur les automates (fonction `check`), puis le fichier `.c` de sortie est généré (fonction `generate_c_file`).

La fonction `read_lea_file`, qui lit les fichiers `.lea` en entrée, est implémentée en utilisant à la fois Flex (fichier `lexer.l`) et Bison (fichier `parser.yxx`). Plus précisément, Flex lit chaque fichier et les découpe en une suite de jetons (ou *tokens*, définis dans `parser.yxx`) qui correspondent à des expressions rationnelles. Ensuite, Bison analyse la séquence de jetons en fonction de la grammaire donnée, et construit les automates correspondant.

## 3 Travail à réaliser

***a.*** **Déterminiser les automates.** Implémentez la fonction `automaton automaton::determine()` dans le fichier `automaton.cpp`. Pour un automate `a`, `a.determine()` retourne un automate déterministe qui reconnaît le même langage que `a`. La fonction de déterminisation est déjà appelée dans le fichier `parser.yxx`, vous n'avez pas à vous en soucier.

***b.*** **Reconnaître les expressions rationnelles.** Modifiez la grammaire du fichier `parser.yxx` pour qu'elle reconnaisse les expressions rationnelles. Pour cela, ajoutez un corps à la règle `rationnal_rules`, en vous inspirant de l'exercice 4.2 du TP 4.

# LEA

Generated by Doxygen 1.8.13

# Contents

# Chapter 1

# Namespace Index

## 1.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

# Chapter 2

# Class Index

## 2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

# Chapter 3

# File Index

## 3.1 File List

Here is a list of all documented files with brief descriptions:

# Chapter 4

# Namespace Documentation

## 4.1 univ_nantes Namespace Reference

### Classes

- struct automaton

  *Encodes non-deterministic finite state automata whose states are integers, and transitions are labelled by characters.*
- class set

  *Encodes a type to manipulate sets algebrically.*
- struct transition

  *Encodes transitions in an automaton.*

### 4.1.1 Detailed Description

Protects all definitions in the LEA project

# Chapter 5

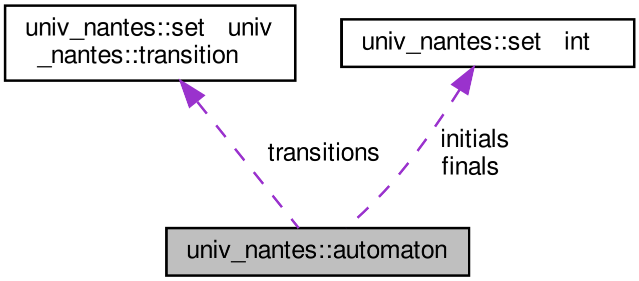
# Class Documentation

## 5.1 univ_nantes::automaton Struct Reference

Encodes non-deterministic finite state automata whose states are integers, and transitions are labelled by characters.

`#include <automaton.hpp>`

Collaboration diagram for univ_nantes::automaton:



### Public Member Functions

- set< int > get_states () const

  *Gets the set of states of the automaton.*
- set< char > get_alphabet () const

  *Gets the set of terminal symbols of the automaton.*
- set< int > epsilon_accessible (set< int > from) const

  *Gets the set of states accessible by following epsilon transitions.*
- set< int > accessible (set< int > from, char c) const

  *Gets the set of states accessible by following one transition.*

- automaton determine () const

  *Gets a new deterministic automaton that recognizes the same language.*
- bool is_deterministic () const

  *Gets whether the automaton is deterministic or not.*
- bool operator== (const automaton &a) const

  *Compares two automata.*

### Public Attributes

- std::string name
- set< int > initials
- set< int > finals
- set< transition > transitions

### Friends

- std::ostream & operator<< (std::ostream &out, const automaton &a)

  *Inserts the description of the automaton a into out.*

## 5.1.1 Detailed Description

Encodes non-deterministic finite state automata whose states are integers, and transitions are labelled by characters.

## 5.1.2 Member Function Documentation

### 5.1.2.1 accessible()

```
set< int > univ_nantes::automaton::accessible (
            set< int > from,
            char c ) const
```

Gets the set of states accessible by following one transition.

Gets the set of states accessible from some state in from by following one transition labeled by c

A state y is included in the returned set if, and only if, there exists a state x in from such that x |-c-> y is contained in transitions.

Example : a.epsilon_accessible(a.accessible(a.epsilon_accessible({1,2}),'a')) returns all states accessible in a, from states 1 or 2, through the word "a".

**Parameters**

| | |
|---|---|
| *from* | a set of states of the automaton |
| *c* | a terminal character in the alphabet of the automaton |

**Returns**

the set of all states accessible from some state in from, following a unique transition

A state y is included in the returned set if, and only if, there exists a state x in from such that x |-c-> y is contained in transitions.

Example : a.epsilon_accessible(a.accessible(a.epsilon_accessible({1,2}),'a')) returns all states accessible in a, from states 1 or 2, through the word "a".

#### 5.1.2.2 determine()

```
automaton univ_nantes::automaton::determine ( ) const
```

Gets a new deterministic automaton that recognizes the same language.

**Returns**

a deterministic automaton

TODO: Question 1. Complete the given implementation of this function!

#### 5.1.2.3 epsilon_accessible()

```
set< int > univ_nantes::automaton::epsilon_accessible (
             set< int > from ) const
```

Gets the set of states accessible by following epsilon transitions.

Gets the set of states accessible from some state in from by following epsilon transitions

A state y is included in the returned set if, and only if, there exists a state x in from and a sequence of states x0=x, x1, ..., xn = y such that each epsilon-transition xi |–> x(i+1) is contained in transitions.

Example : a.epsilon_accessible(a.initials) returns all states accessible in a, through the empty word.

**Parameters**

| | |
|---|---|
| *from* | a set of states of the automaton |

**Returns**

the set of all states epsilon-accessible from some state in from

A state y is included in the returned set if, and only if, there exists a state x in from and a sequence of states x0=x, x1, ..., xn = y such that each epsilon-transition xi |–> x(i+1) is contained in transitions.

Example : a.epsilon_accessible(a.initials) returns all states accessible in a, through the empty word.

#### 5.1.2.4 get_alphabet()

```
set< char > univ_nantes::automaton::get_alphabet ( ) const
```

Gets the set of terminal symbols of the automaton.

**Returns**

the set of lower-case letters that label at least one transition

#### 5.1.2.5 get_states()

```
set< int > univ_nantes::automaton::get_states ( ) const
```

Gets the set of states of the automaton.

**Returns**

a set of states, including all states accessible from the initial states

A state is contained in the set returned if it is contained in initials, finals, or at the start or and of any transition.

#### 5.1.2.6 is_deterministic()

```
bool univ_nantes::automaton::is_deterministic ( ) const
```

Gets whether the automaton is deterministic or not.

**Returns**

true if the automaton is deterministic, false otherwise

An automaton is considered to be deterministic if, and only if, it has exactly one initial state, no epsilon-transition, and no two transitions starting in the same state and ending in a different states, with a different label.

#### 5.1.2.7 operator==()

```
bool univ_nantes::automaton::operator== (
            const automaton & a ) const  [inline]
```

Compares two automata.

**Parameters**

| | |
|---|---|
| *a* | an automaton with which to compare ∗this |

**Returns**

true if ∗this and a are the same automata, false otherwise

Two automata are considered equal if they have the same name and sets of initial and final states, and transitions. Set equality is defined by double inclusion (see univ_nantes::set).

### 5.1.3 Friends And Related Function Documentation

#### 5.1.3.1 operator<<

```
std::ostream & operator<< (
             std::ostream & out,
             const automaton & a )  [friend]
```

Inserts the description of the automaton a into out.

**Parameters**

| | |
|---|---|
| *out* | ostream object where the automaton is inserted. |
| *a* | automaton object with the content to insert. |

**Returns**

The same as parameter out.

The format under which the automaton is displayed is the same as the input format expected in a .lea file

### 5.1.4 Member Data Documentation

#### 5.1.4.1 finals

```
set<int> univ_nantes::automaton::finals
```

Set of final (a.k.a. accepting) states of the automaton.

#### 5.1.4.2 initials

```
set<int> univ_nantes::automaton::initials
```

Set of initial states of the automaton.

#### 5.1.4.3 name

```
std::string univ_nantes::automaton::name
```

Name of the automaton, or of the language recognized by the automaton.

#### 5.1.4.4 transitions

```
set<transition> univ_nantes::automaton::transitions
```

Set of transitions of the automaton.

The documentation for this struct was generated from the following files:

- automaton.hpp
- automaton.cpp

## 5.2 univ_nantes::set< T > Class Template Reference

Encodes a type to manipulate sets algebrically.

```
#include <set.hpp>
```

### Public Member Functions

- set ()

  *Default constructor.*
- set (const set &e)

  *copy constructor*
- set (const std::initializer_list< T > &c)

  *Initializer list constructor.*
- ∼set ()

  *Default destructor.*
- std::size_t size () const

  *Gets the number of elements in the set.*
- bool contains (const T &x) const

  *Gets whether an element is contained in the set.*
- const set & operator= (const set &e)

  *assignation operator*
- const T & operator[ ] (std::size_t i) const

  *Gives a read-only access to elements in the set.*
- set & operator|= (const set &rhs)

  *Inserts all elements of rhs into the set.*
- set & operator|= (const T &rhs)

  *Inserts rhs into the set.*
- set & **operator &=** (const set &rhs)
- set & **operator &=** (const T &rhs)
- set & operator-= (const set &rhs)

  *Removes from the set all elements in rhs.*
- set & operator-= (const T &rhs)

  *Removes rhs from the set.*
- auto begin () const

  *Returns an iterator pointing to the first element in the set.*
- auto end () const

  *Returns an iterator referring to the past-the-end element in the set.*

### Friends

- std::ostream & operator<< (std::ostream &out, const set &e)

  *Inserts the description of the set e into out.*
- set operator| (set lhs, const set &rhs)

  *Computes the union between lhs and rhs.*
- set operator| (set lhs, const T &rhs)

  *Computes the union between lhs and the set only containing rhs.*
- set **operator &** (set lhs, const set &rhs)
- set **operator &** (set lhs, const T &rhs)
- set operator- (set lhs, const set &rhs)

  *Computes the set difference between lhs and rhs.*
- set operator- (set lhs, const T &rhs)

  *returns the set difference of lhs and {rhs}*
- bool operator<= (const set &lhs, const set &rhs)

  *compares whether lhs is included into, or equal to, rhs*
- bool operator== (const set &lhs, const set &rhs)

  *compares whether lhs and rhs contain the same elements*
- bool operator!= (const set &lhs, const set &rhs)

  *compares whether lhs and rhs contain different elements*
- bool operator>= (const set &lhs, const set &rhs)

  *compares whether rhs is included into, or equal to, lhs*
- bool operator< (const set &lhs, const set &rhs)

  *compares whether lhs is strictly included into rhs*
- bool operator> (const set &lhs, const set &rhs)

  *compares whether rhs is strictly included into lhs*

### 5.2.1 Detailed Description

**template<typename T>**
**class univ_nantes::set< T >**

Encodes a type to manipulate sets algebrically.

This class defines the operators | for the union, & for the intersection, and - for the set difference. Let S1 and S2 be sets. In the following examples, we define set<int> S1 = {1, 2, 3}; set<int> S2 = {3, 4, 5};

- set union: S1 | S2 represents the union of S1 and S2. For example, S1 | S2 == {1, 2, 3, 4, 5}, and S1 |= S2 is the same as S1 = {1, 2, 3, 4, 5};
- set intersection: S1 & S2 represents the intersection of S1 and S2. For example, S1 | S2 == {1, 2, 3, 4, 5}, and S1 |= S2 is the same as S1 = {1, 2, 3, 4, 5};

### 5.2.2 Constructor & Destructor Documentation

#### 5.2.2.1 set() [1/3]

```
template<typename T>
univ_nantes::set< T >::set ( )  [inline]
```

Default constructor.

creates an empty set

#### 5.2.2.2 set() [2/3]

```
template<typename T>
univ_nantes::set< T >::set (
             const set< T > & e )  [inline]
```

copy constructor

**Parameters**

|   |   |
|---|---|
| *e* | a set to copy |

Allows to write set<T> s = e;

#### 5.2.2.3 set() [3/3]

```
template<typename T>
univ_nantes::set< T >::set (
             const std::initializer_list< T > & c )  [inline]
```

Initializer list constructor.

**Parameters**

|   |   |
|---|---|
| *c* | the initializer list |

Allows to write set<T> s = {1, 2, 3, 4};

### 5.2.3 Member Function Documentation

#### 5.2.3.1 begin()

```
template<typename T>
auto univ_nantes::set< T >::begin ( ) const  [inline]
```

Returns an iterator pointing to the first element in the set.

**Returns**

e An iterator to the beginning of the container.

Necessary to write : for(T x : s), where s is of type set<T>

#### 5.2.3.2 contains()

```
template<typename T>
bool univ_nantes::set< T >::contains (
             const T & x ) const  [inline]
```

Gets whether an element is contained in the set.

bool contains(const T& x) const

**Parameters**

| | |
|---|---|
| *x* | an element that may be contained in the set or not |

**Returns**

true if x is in the set, false otherwise

#### 5.2.3.3 end()

```
template<typename T>
auto univ_nantes::set< T >::end ( ) const  [inline]
```

Returns an iterator referring to the past-the-end element in the set.

**Returns**

e An iterator to the element past the end of the sequence.

Necessary to write : for(T x : s), where s is of type set<T>

#### 5.2.3.4 operator-=() [1/2]

```
template<typename T>
set & univ_nantes::set< T >::operator-= (
             const set< T > & rhs )  [inline]
```

Removes from the set all elements in rhs.

**Parameters**

| | |
|---|---|
| *rhs* | A set of elements to be removed |

**Returns**

The resulting set

Allows to write s1 -= s2, interpreted as "s1 = s1 minus s2"

#### 5.2.3.5 operator-=() [2/2]

```
template<typename T>
set & univ_nantes::set< T >::operator-= (
            const T & rhs )  [inline]
```

Removes rhs from the set.

**Parameters**

| | |
|---|---|
| *rhs* | A unique element to remove |

**Returns**

The resulting set

Allows to write s1 -= x, interpreted as "s1 = s1 minus {x}".

#### 5.2.3.6 operator=()

```
template<typename T>
const set & univ_nantes::set< T >::operator= (
            const set< T > & e )  [inline]
```

assignation operator

**Parameters**

| | |
|---|---|
| *e* | a set to copy |

**Returns**

the same set

Allows to write s = e;

#### 5.2.3.7 operator[]()

```
template<typename T>
const T & univ_nantes::set< T >::operator[] (
             std::size_t i ) const  [inline]
```

Gives a read-only access to elements in the set.

**Parameters**

| | |
|---|---|
| *i* | An integer index between 0 and size()-1 |

**Returns**

Some element in the set

Allows to write for(int i = 0; i<s.size(); ++i) cout << s[i];

If i and j are valid indices,

- s[i]==s[j] must return true if, and only if, i==j;
- s.contains(s[i]) must return true.

#### 5.2.3.8 operator" | =() [1/2]

```
template<typename T>
set & univ_nantes::set< T >::operator|= (
             const set< T > & rhs )  [inline]
```

Inserts all elements of rhs into the set.

**Parameters**

| | |
|---|---|
| *rhs* | A set of elements to be inserted |

**Returns**

The resulting set

Allows to write s1 |= s2, interpreted as "s1 = s1 union s2"

#### 5.2.3.9 operator" | =() [2/2]

```
template<typename T>
set & univ_nantes::set< T >::operator|= (
             const T & rhs )  [inline]
```

Inserts rhs into the set.

**Parameters**

| | |
|---|---|
| *rhs* | A unique element to be inserted |

**Returns**

The resulting set

Allows to write s1 |= x, interpreted as "s1 = s1 union {x}".

#### 5.2.3.10 size()

```
template<typename T>
std::size_t univ_nantes::set< T >::size ( ) const  [inline]
```

Gets the number of elements in the set.

std::size_t size() const

**Returns**

The number of elements contained in the set

### 5.2.4 Friends And Related Function Documentation

#### 5.2.4.1 operator"!=

```
template<typename T>
bool operator!= (
            const set< T > & lhs,
            const set< T > & rhs )  [friend]
```

compares whether lhs and rhs contain different elements

**Parameters**

| | |
|---|---|
| *lhs* | the left hand side set in the comparison |
| *rhs* | the right hand side set in the comparison |

**Returns**

true if lhs and rhs contain different elements; false otherwise

Allows to write s1 != s2, interpreted as "s1 not equals to s2".

#### 5.2.4.2 operator- [1/2]

```
template<typename T>
set operator- (
            set< T > lhs,
            const set< T > & rhs )  [friend]
```

Computes the set difference between lhs and rhs.

**Parameters**

| | |
|---|---|
| *lhs* | the first set in the set difference |
| *rhs* | the second set in the set difference |

**Returns**

The set difference of lhs and rhs

Allows to write s1 - s2, interpreted as "s1 minus s2".

#### 5.2.4.3 operator- [2/2]

```
template<typename T>
set operator- (
            set< T > lhs,
            const T & rhs )  [friend]
```

returns the set difference of lhs and {rhs}

**Parameters**

| | |
|---|---|
| *lhs* | the first set in the set difference |
| *rhs* | A unique element to remove in the set difference |

**Returns**

The resulting set

Allows to write s1 - x, interpreted as "s1 = s1 minus {x}".

#### 5.2.4.4 operator<

```
template<typename T>
bool operator< (
            const set< T > & lhs,
            const set< T > & rhs )  [friend]
```

compares whether lhs is strictly included into rhs

**Parameters**

| | |
|---|---|
| *lhs* | the left hand side set in the comparison |
| *rhs* | the right hand side set in the comparison |

**Returns**

true if lhs is strictly included into rhs; false otherwise

Allows to write s1 < s2, interpreted as "s1 strictly included into s2".

#### 5.2.4.5 operator<<

```
template<typename T>
std::ostream & operator<< (
            std::ostream & out,
            const set< T > & e )  [friend]
```

Inserts the description of the set e into out.

**Parameters**

| | |
|---|---|
| *out* | ostream object where the set is inserted. |
| *e* | set object with the content to insert. |

**Returns**

The same as parameter out.

The set is displayed as an enumeration between braces, e.g. "{1, 2, 3}".

#### 5.2.4.6 operator<=

```
template<typename T>
bool operator<= (
            const set< T > & lhs,
            const set< T > & rhs )  [friend]
```

compares whether lhs is included into, or equal to, rhs

**Parameters**

| | |
|---|---|
| *lhs* | the left hand side set in the comparison |
| *rhs* | the right hand side set in the comparison |

**Returns**

true if lhs in included into, or equal to, rhs; false otherwise

Allows to write s1 <= s2, interpreted as "s1 included into s2".

#### 5.2.4.7 operator==

```
template<typename T>
bool operator== (
            const set< T > & lhs,
            const set< T > & rhs )  [friend]
```

compares whether lhs and rhs contain the same elements

**Parameters**

| | |
|---|---|
| *lhs* | the left hand side set in the comparison |
| *rhs* | the right hand side set in the comparison |

**Returns**

true if lhs and rhs contain the same elements; false otherwise

Allows to write s1 == s2, interpreted as "s1 equals s2".

#### 5.2.4.8 operator>

```
template<typename T>
bool operator> (
            const set< T > & lhs,
            const set< T > & rhs )  [friend]
```

compares whether rhs is strictly included into lhs

**Parameters**

| | |
|---|---|
| *lhs* | the left hand side set in the comparison |
| *rhs* | the right hand side set in the comparison |

**Returns**

true if rhs is strictly included into rhs; false otherwise

Allows to write s1 > s2, interpreted as "s2 strictly included into s1".

#### 5.2.4.9 operator>=

```
template<typename T>
bool operator>= (
            const set< T > & lhs,
            const set< T > & rhs )  [friend]
```

compares whether rhs is included into, or equal to, lhs

**Parameters**

| | |
|---|---|
| *lhs* | the left hand side set in the comparison |
| *rhs* | the right hand side set in the comparison |

**Returns**

true if rhs in included into, or equal to, lhs; false otherwise

Allows to write s1 >= s2, interpreted as "s2 included into s1".

### 5.2.4.10 operator" | [1/2]

```
template<typename T>
set operator| (
            set< T > lhs,
            const set< T > & rhs )  [friend]
```

Computes the union between lhs and rhs.

**Parameters**

| | |
|---|---|
| *lhs* | the first set in the union |
| *rhs* | the second set in the union |

**Returns**

The union of lhs and rhs

Allows to write s1 | s2, interpreted as "s1 union s2".

### 5.2.4.11 operator" | [2/2]

```
template<typename T>
set operator| (
            set< T > lhs,
            const T & rhs )  [friend]
```

Computes the union between lhs and the set only containing rhs.

**Parameters**

| | |
|---|---|
| *lhs* | the first set in the union |
| *rhs* | the value to add |

**Returns**

The union of lhs and rhs

Allows to write s1 | x, interpreted as "s1 union {x}".

The documentation for this class was generated from the following file:

- set.hpp

## 5.3 univ_nantes::transition Struct Reference

Encodes transitions in an automaton.

`#include <automaton.hpp>`

### Public Member Functions

- transition (int s, char t, int e)

  *constructs a new transition*
- transition (int s, int e)

  *constructs a new epsilon transition*
- transition ()

  *default constructor*
- bool is_epsilon () const

  *determines whether the transition is an epsilon transition or not*
- bool operator== (const transition &t) const

  *Compares two transitions.*

### Public Attributes

- int start
- char terminal
- int end

### Friends

- std::ostream & operator<< (std::ostream &out, const transition &t)

  *Inserts the description of the transition t into out.*

### 5.3.1 Detailed Description

Encodes transitions in an automaton.

A transition is encoded as a tripple (start, terminal, end). Epsilon transitions are encoded as transitions with the character '\0' as terminal

### 5.3.2 Constructor & Destructor Documentation

#### 5.3.2.1 transition() [1/2]

```
univ_nantes::transition::transition (
            int s,
            char t,
            int e )  [inline]
```

constructs a new transition

**Parameters**

| | |
|---|---|
| *s* | State in which the transition can be activated |
| *t* | Label of the transition |
| *e* | State in which the transition leads after it has been activated |

#### 5.3.2.2 transition() [2/2]

```
univ_nantes::transition::transition (
             int s,
             int e )  [inline]
```

constructs a new epsilon transition

**Parameters**

| | |
|---|---|
| *s* | State in which the transition can be activated |
| *e* | State in which the transition leads after it has been activated |

### 5.3.3 Member Function Documentation

#### 5.3.3.1 is_epsilon()

```
bool univ_nantes::transition::is_epsilon ( ) const  [inline]
```

determines whether the transition is an epsilon transition or not

**Returns**

true if called on an epsilon transition, false otherwise

#### 5.3.3.2 operator==()

```
bool univ_nantes::transition::operator== (
             const transition & t ) const  [inline]
```

Compares two transitions.

**Parameters**

| | |
|---|---|
| *t* | a transition with which to compare ∗this |

**Returns**

true if ∗this and t are the same transition, false otherwise

Two transitions are considered equal if they have the same start and end states, and they are labelled by the same terminal.

## 5.3.4 Friends And Related Function Documentation

### 5.3.4.1 operator<<

```
std::ostream & operator<< (
            std::ostream & out,
            const transition & t )  [friend]
```

Inserts the description of the transition t into out.

**Parameters**

| | |
|---|---|
| *out* | ostream object where the transition is inserted. |
| *t* | transition object with the content to insert. |

**Returns**

The same as parameter out.

Examples: Textual display of a transition t(0, 'a', 1) is "0 |-a-> 1"

Textual display of an epsilon transition t(0, 1) is "0 |--> 1"

## 5.3.5 Member Data Documentation

### 5.3.5.1 end

```
int univ_nantes::transition::end
```

State in which the transition leads after it has been activated.

### 5.3.5.2 start

```
int univ_nantes::transition::start
```

State in which the transition can be activated.

#### 5.3.5.3 terminal

```
char univ_nantes::transition::terminal
```

Label of the transition: the transition can be activated only when terminal is read.

The documentation for this struct was generated from the following file:

- automaton.hpp

# Chapter 6

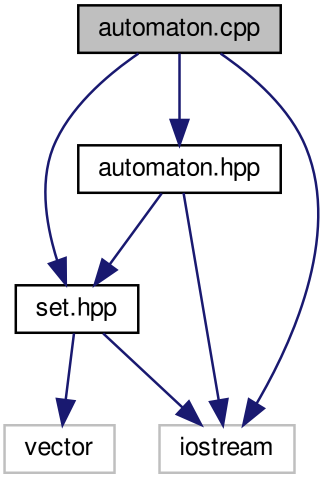
# File Documentation

## 6.1 automaton.cpp File Reference

Implementation file containing the code for the functions that could not be implemented in "automaton.hpp".

```
#include "automaton.hpp"
#include "set.hpp"
#include <iostream>
```

Include dependency graph for automaton.cpp:



### 6.1.1 Detailed Description

Implementation file containing the code for the functions that could not be implemented in "automaton.hpp".

**Author**

Matthieu Perrin

**Version**

1

**Date**

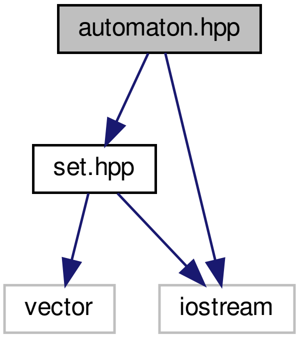
11-16-2020

## 6.2 automaton.hpp File Reference

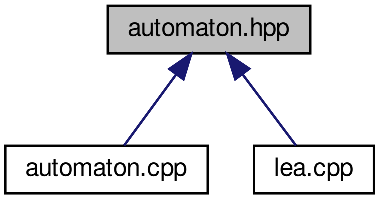
Header file containing the declaration of the transition and automaton types.

```
#include "set.hpp"
#include <iostream>
```

Include dependency graph for automaton.hpp:



This graph shows which files directly or indirectly include this file:



## Classes

- struct univ_nantes::transition

  *Encodes transitions in an automaton.*
- struct univ_nantes::automaton

  *Encodes non-deterministic finite state automata whose states are integers, and transitions are labelled by characters.*

## Namespaces

- univ_nantes

### 6.2.1 Detailed Description

Header file containing the declaration of the transition and automaton types.

Header file containing the declaration of the set type.

**Author**

Matthieu Perrin

**Version**

1

**Date**

11-16-2020

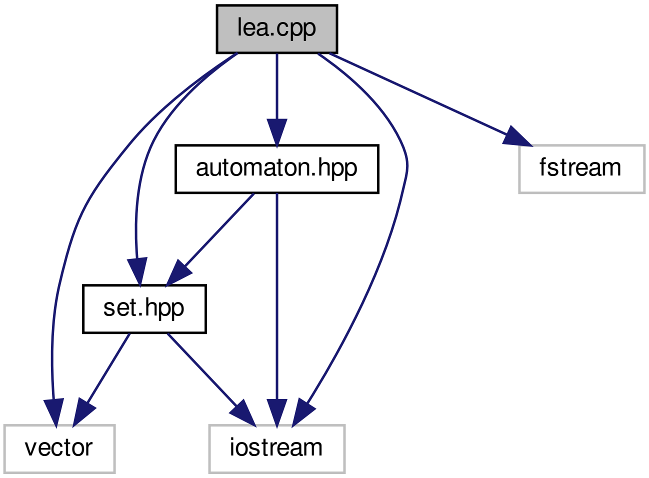
## 6.3 lea.cpp File Reference

File containing the main function of the program.

```
#include "automaton.hpp"
#include "set.hpp"
#include <vector>
#include <iostream>
```

```
#include <fstream>
```

Include dependency graph for lea.cpp:



## Functions

- set< automaton > read_lea_file (char ∗file)

  *Parses the lea file whose name is file.*
- bool check (set< automaton > automata)

  *Checks that the automata are well-formed to generate the output file.*
- void generate_c_file (ostream &out, set< automaton > automata)

  *Generates the c code executing the given automata.*
- int main (int argc, char ∗argv[ ])

  *Starts the program.*

### 6.3.1 Detailed Description

File containing the main function of the program.

**Author**

Matthieu Perrin

**Version**

1

**Date**

11-16-2020

### 6.3.2 Function Documentation

#### 6.3.2.1 check()

```
bool check (
             set< automaton > automata )
```

Checks that the automata are well-formed to generate the output file.

**Parameters**

| | |
|---|---|
| *automata* | the set of automata to check |

**Returns**

true if everything is correct

This functions does the following checks: 1) There is at least one automaton 2) all automata are deterministic 3) all automata have a name 4) all automata have a different name

#### 6.3.2.2 generate_c_file()

```
void generate_c_file (
             ostream & out,
             set< automaton > automata )
```

Generates the c code executing the given automata.

**Parameters**

| | |
|---|---|
| *out* | the stream in which the c code must be included |
| *automata* | the set of automata to print |

#### 6.3.2.3 main()

```
int main (
             int argc,
             char * argv[] )
```

Starts the program.

**Parameters**

| | |
|---|---|
| *argc* | number of arguments in the command line |
| *argv* | arguments in the command line |

**Returns**

a potential error code

#### 6.3.2.4 read_lea_file()

```
set< automaton > read_lea_file (
            char * file )
```

Parses the lea file whose name is file.

**Parameters**

| | |
|---|---|
| *file* | the name of the input file |

**Returns**

a set of automata, recognizing languages in the input file

This functions is implemented in File parser.yxx

# Index