<presentation>

<title>Programmation XML: de la théorie aux outils</title>

_{24ème Journées Frankofônes des Langages Applicatifs}

<date>3-6 Février 2013, Aussois</date>

<author>**Kim Nguyễn**</author>

<email>kn@lri.fr</email>

</presentation>

<title>But de ce cours</title>

Réciter 1400 pages de spécifications écrites par le W3C

<title>But de ce cours</title>

~~Réciter 1400 pages de spécifications écrites par le W3C~~

Donner un bref apperçu d'XML

<title>But de ce cours</title>

~~Réciter 1400 pages de spécifications écrites par le W3C~~

Donner un bref apperçu d'XML

Décrire quelques concepts théoriques

<title>But de ce cours</title>

~~Réciter 1400 pages de spécifications écrites par le W3C~~

Donner un bref apperçu d'XML

Décrire quelques concepts théoriques

Présenter des outils issus de la recherche

<title>But de ce cours</title>

~~Réciter 1400 pages de spécifications écrites par le W3C~~

Donner un bref apperçu d'XML

Décrire quelques concepts théoriques

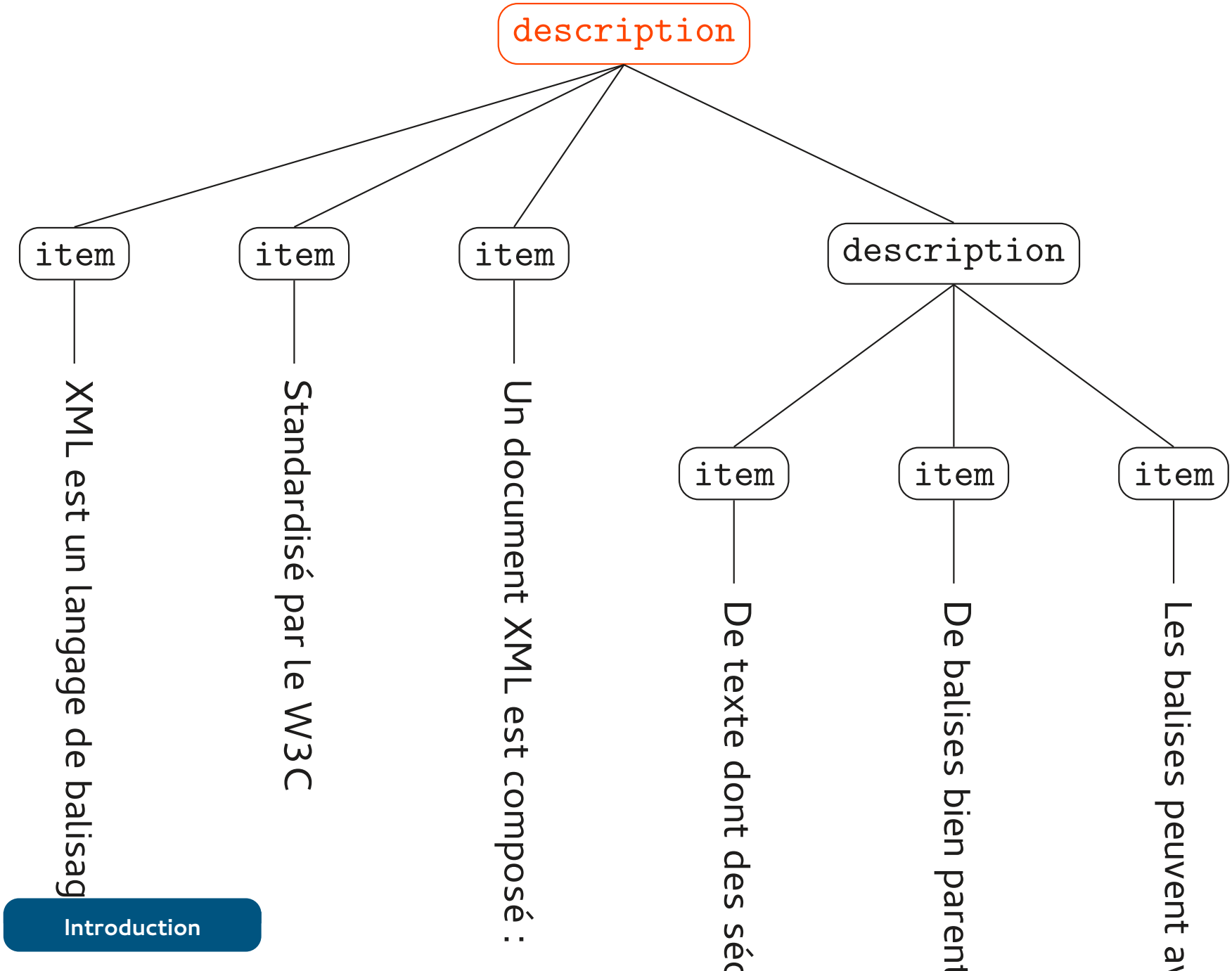
Présenter des outils issus de la recherche

$\Rightarrow$ favoriser la réutilisation dans d'autres domaines (typeurs, solveurs SMT , *model checking* . . .)

<title>XML ?</title>

```
<description>
 <item>XML est un langage de balisage</item>
 <item>Standardisé par le W3C</item>
 <item>Un document XML est composé</item>
 <description>
  <item>De texte dont des séquences d&apos;échappement</item>
  <item>De balises, bien parenthésées (dont une racine)</item>
  <item id="id1">Les balises peuvent avoir des attributs</item>
 </description>
</description>
```

<title>Document XML = Arbre</title>



<title>Utilisations</title>

- Données semi-structurées (par opositions aux tables relationnelles)
- Représentation arborescente de données
- Utilisations multiples :

  Standards du web XHTML, Atom, RSS, SOAP, SVG, MathML, ...
  Documents OpenDocument format
  Bio-informatique UniProt XML, SBML
  Finance XBRL
  Linguistique Treebank, PAULA XML, ...

<title>Quels avantages ?</title>

1. Standardisé par le W3C
2. Cadre commun pour la sérialisation et l'échange de données
3. Bibliothèques de *parsing* génériques
4. Possiblité de *spécifier* les documents
5. Langage(s) standards de manipulation de documents (XSLT, XPath, . . .)
6. *Mapping* standardisé entre document XML et langages objets (DOM)

<title>Quels inconvénients ?</title>

1. Standardisé par le W3C
2. Verbeux
3. DOM est mal adapté aux langages fonctionnels
4. Spécifications très informelles

<title>Au programme</title>

Pratique

1. Validation de documents

3. ℂDuce et le sous-typage sémantique

4. XPath, XQuery, encodage d'arbres par intervalles

5. Logiques pour les arbres

2. Automates d'arbres

Théorie

1. Validation de documents

<title>Contraintes sur les documents</title>

On veut pouvoir définir des contraintes sur un document :

1. ensemble des noms de balises autorisés
2. imbrication et contenu des balises
3. (nom et valeurs des attributs)
4. (« forme » du texte : date, nombre, . . .)

Il y a plusieurs standards pour cela :

1. Document Type Definition (DTD)
2. (XML Schema)
3. (RelaxNG)

<title>Document Type Definition</title>

1. Définit le contenu de chaque balise par une expression régulière
2. Ne permet pas d'imposer des contraintes sur le texte

Exemple :

```
<!ELEMENT description      (item | description)+ >
<!ELEMENT item             (#PCDATA | b | i )* >
<!ELEMENT i                (#PCDATA | b | i )* >
<!ELEMENT b                (#PCDATA | b | i )* >
```

<title>Document Type Definition</title>

1. Définit le contenu de chaque balise par une expression régulière
2. Ne permet pas d'imposer des contraintes sur le texte

Exemple :

```
<!ELEMENT description     (item | description)+ >
<!ELEMENT item            (#PCDATA | b | i )* >
<!ELEMENT i               (#PCDATA | b | i )* >
<!ELEMENT b               (#PCDATA | b | i )* >
```

Contrainte : le document doit être vérifiable en *streaming*
Remarque : on s'autorise une pile proportionnelle à la profondeur de l'arbre

<title>Exemple de validation en streaming</title>

```
<!ELEMENT description      (item | description)+ >
<!ELEMENT item             (#PCDATA | b | i )*>
<!ELEMENT i                (#PCDATA | b | i )* >
<!ELEMENT b                (#PCDATA | b | i )* >
```

<title>Exemple de validation en streaming</title>

```
<!ELEMENT description      (item | description)+ >
<!ELEMENT item             (#PCDATA | b | i )*>
<!ELEMENT i                (#PCDATA | b | i )* >
<!ELEMENT b                (#PCDATA | b | i )* >
```

Le parseur renvoie la séquence d'évènements :

```
<description>
```

<title>Exemple de validation en streaming</title>

```
<!ELEMENT description      (item | description)+ >
<!ELEMENT item             (#PCDATA | b | i )*>
<!ELEMENT i                (#PCDATA | b | i )* >
<!ELEMENT b                (#PCDATA | b | i )* >
```

Le parseur renvoie la séquence d'évènements :

```
<description>
   <item>
```

<title>Exemple de validation en streaming</title>

```
<!ELEMENT description      (item | description)+ >
<!ELEMENT item             (#PCDATA | b | i )*>
<!ELEMENT i                (#PCDATA | b | i )* >
<!ELEMENT b                (#PCDATA | b | i )* >
```

Le parseur renvoie la séquence d'évènements :

```
<description>
   <item>
 Blah blah
```

<title>Exemple de validation en streaming</title>

```
<!ELEMENT description      (item | description)+ >
<!ELEMENT item             (#PCDATA | b | i )*>
<!ELEMENT i                (#PCDATA | b | i )* >
<!ELEMENT b                (#PCDATA | b | i )* >
```

Le parseur renvoie la séquence d'évènements :

```
<description>
   <item>
 Blah blah
  <b>
```

<title>Exemple de validation en streaming</title>

```
<!ELEMENT description      (item | description)+ >
<!ELEMENT item             (#PCDATA | b | i )*>
<!ELEMENT i                (#PCDATA | b | i )* >
<!ELEMENT b                (#PCDATA | b | i )* >
```

Le parseur renvoie la séquence d'évènements :

```
<description>
   <item>
 Blah blah
  <b>
<description>
```

<title>Exemple de validation en streaming</title>

```
<!ELEMENT description      (item | description)+ >
<!ELEMENT item             (#PCDATA | b | i )*>
<!ELEMENT i                (#PCDATA | b | i )* >
<!ELEMENT b                (#PCDATA | b | i )* >
<!ELEMENT b                description+ >
```

Le parseur renvoie la séquence d'évènements :

```
<description>
   <item>
 Blah blah
  <b>
<description>
```

<title>Exemple de validation en streaming</title>

```
<!ELEMENT description      (item | description)+ >
<!ELEMENT item             (#PCDATA | b | i )*, b>
<!ELEMENT i                (#PCDATA | b | i )* >
<!ELEMENT b                (#PCDATA | b | i )* >
```

Le parseur renvoie la séquence d'évènements :

```
<description>
   <item>
 Blah blah
  <b>
<description>
```

<title>Contraintes sur les DTDs</title>

1. Une seule production par nom de balise
2. Expression 1-nonambiguë *[Brüggemann-Klein, Wood, 97]*

<title>Contraintes sur les DTDs</title>

1. Une seule production par nom de balise
2. Expression 1-nonambiguë *[Brüggemann-Klein, Wood, 97]*

Exemple :

```
(a | b)* a
```

<title>Contraintes sur les DTDs</title>

1. Une seule production par nom de balise
2. Expression 1-nonambiguë *[Brüggemann-Klein, Wood, 97]*

Exemple :

`(a | b)* a` $\rightsquigarrow$ `(`a_1 `|` b_1`)*` a_2

<title>Contraintes sur les DTDs</title>

1. Une seule production par nom de balise
2. Expression 1-nonambiguë *[Brüggemann-Klein, Wood, 97]*

Exemple :

`(a | b)* a` $\rightsquigarrow$ `(a`$_1$ `| b`$_1$`)* a`$_2$

Considérons le mot `baa` :

1. on lit `b`, on sait que c'est `b`$_1$
2. on lit `a`, on ne sait pas si c'est `a`$_1$ ou `a`$_2$

<title>Contraintes sur les DTDs</title>

1. Une seule production par nom de balise
2. Expression 1-nonambiguë *[Brüggemann-Klein, Wood, 97]*

Exemple :

$(a \mid b)^* a \rightsquigarrow (a_1 \mid b_1)^* a_2$

Considérons le mot baa :

1. on lit b, on sait que c'est b_1
2. on lit a, on ne sait pas si c'est a_1 ou a_2

Remarque : $b_1^* a_1 (b_2^* a_2)^*$ dénote le même langage et est nonambiguë

La 1-nonambiguité est une propriété syntaxique

<title>Automate de Glushkov</title>

Construction simple a partir de l'expression régulière :

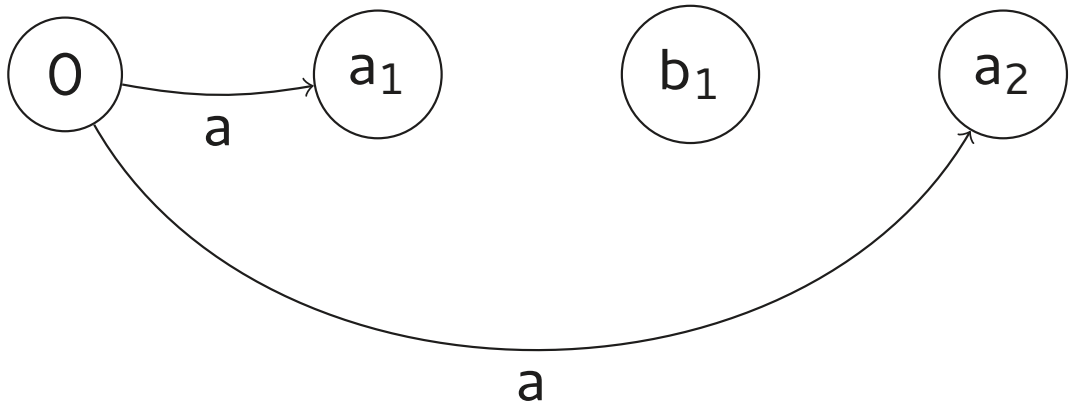
- $(a_1 \mid b_1)^* a_2$

- Automate de Glushkov deterministe $\Leftrightarrow$ expression 1-nonambiguë
- Validation en *streaming* (pile $o(depth(D))$ *[Segoufin, Vianu 2002]*

<title>Automate de Glushkov</title>

Construction simple a partir de l'expression régulière :

- $(a_1 \mid b_1)^* a_2$

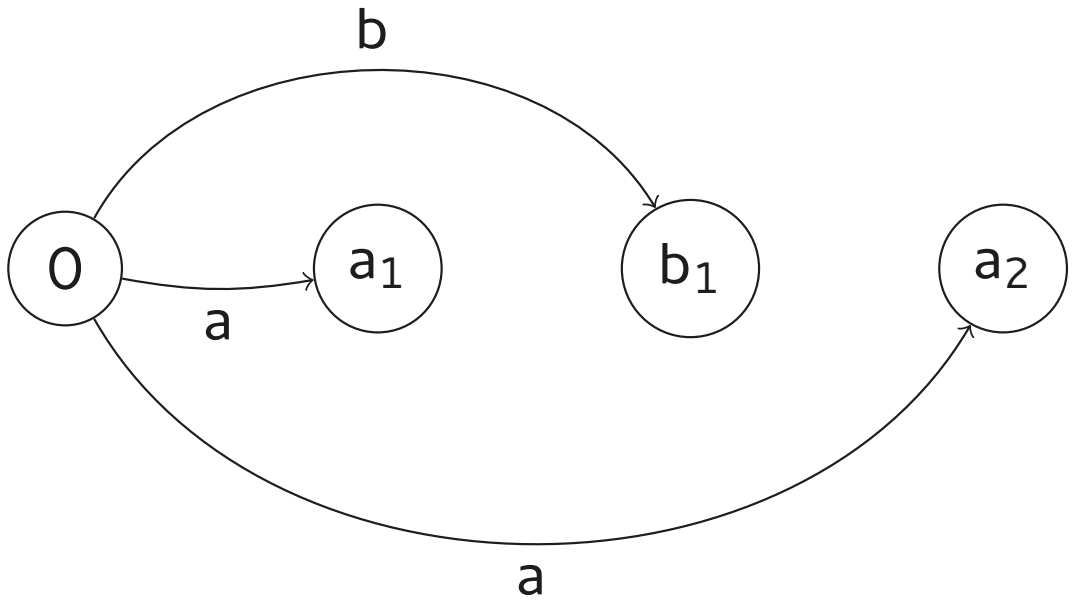


- Automate de Glushkov deterministe $\Leftrightarrow$ expression 1-nonambiguë
- Validation en *streaming* (pile $o(depth(D))$ *[Segoufin, Vianu 2002]*

<title>Automate de Glushkov</title>

Construction simple a partir de l'expression régulière :

- $(a_1 \mid b_1)^* a_2$

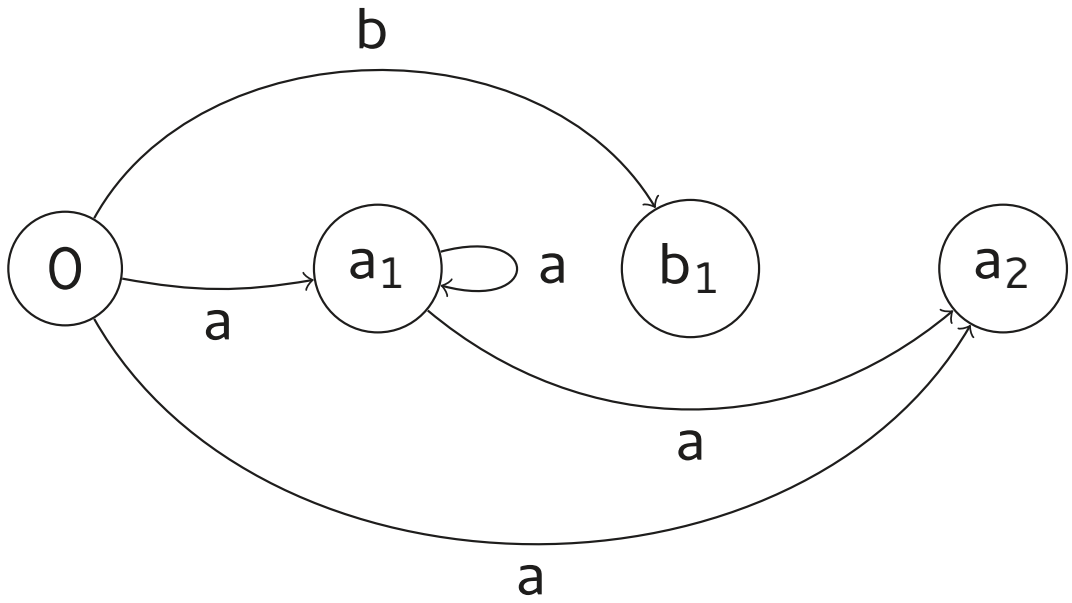


- Automate de Glushkov deterministe $\Leftrightarrow$ expression 1-nonambiguë
- Validation en *streaming* (pile $o(depth(D))$ *[Segoufin, Vianu 2002]*

<title>Automate de Glushkov</title>

Construction simple a partir de l'expression régulière :

- $(a_1 \mid b_1)^* a_2$

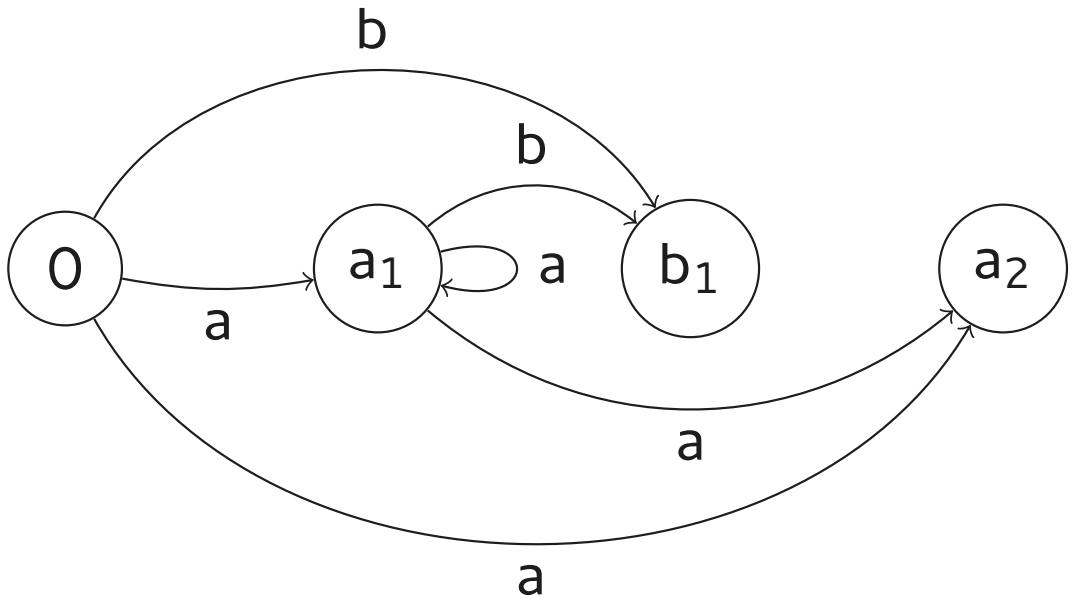


- Automate de Glushkov deterministe $\Leftrightarrow$ expression 1-nonambiguë
- Validation en *streaming* (pile $o(depth(D))$ *[Segoufin, Vianu 2002]*

<title>Automate de Glushkov</title>

Construction simple a partir de l'expression régulière :

- $(a_1 \mid b_1)^* a_2$

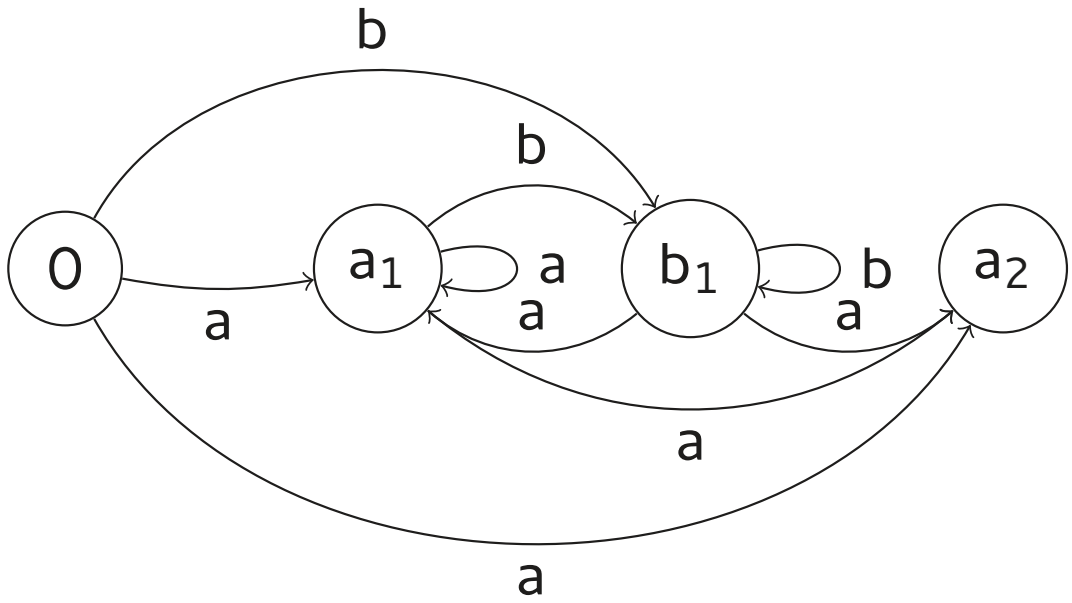


- Automate de Glushkov deterministe $\Leftrightarrow$ expression 1-nonambiguë
- Validation en *streaming* (pile $o(depth(D))$ *[Segoufin, Vianu 2002]*

<title>Automate de Glushkov</title>

Construction simple a partir de l'expression régulière :

- $(a_1 \mid b_1)^* a_2$

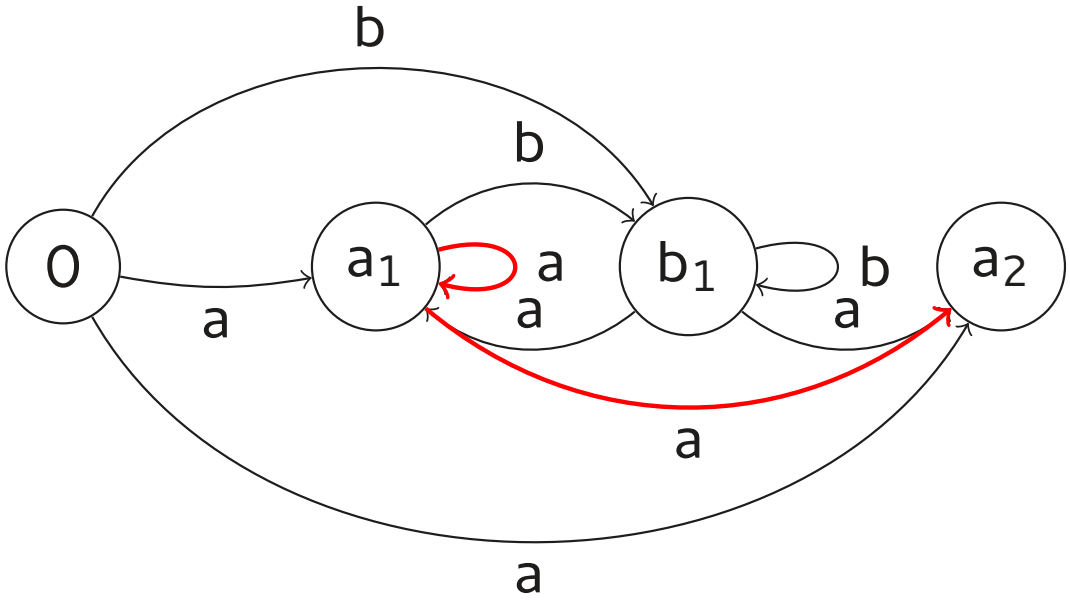


- Automate de Glushkov deterministe $\Leftrightarrow$ expression 1-nonambiguë
- Validation en *streaming* (pile $o(depth(D))$ *[Segoufin, Vianu 2002]*

<title>Automate de Glushkov</title>

Construction simple a partir de l'expression régulière :

- $(a_1 \mid b_1)^* a_2$

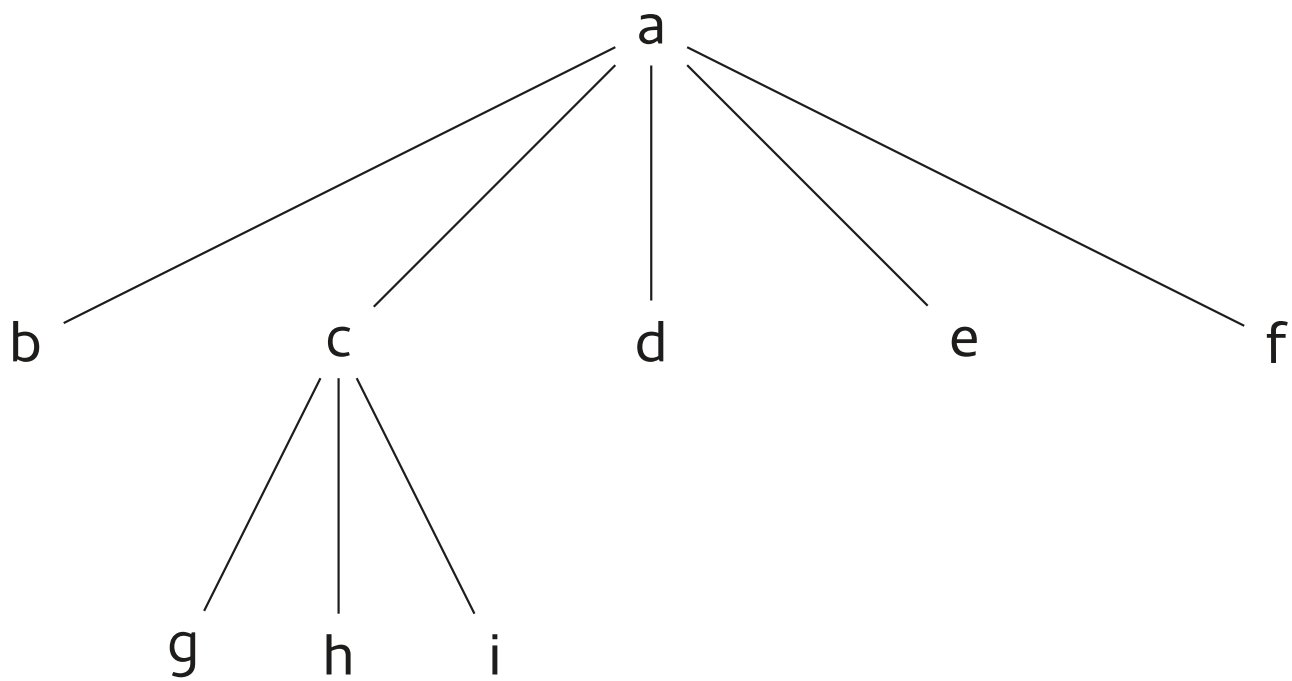


- Automate de Glushkov deterministe $\Leftrightarrow$ expression 1-nonambiguë
- Validation en *streaming* (pile $o(depth(D))$ *[Segoufin, Vianu 2002]*

2. Automates d’arbres

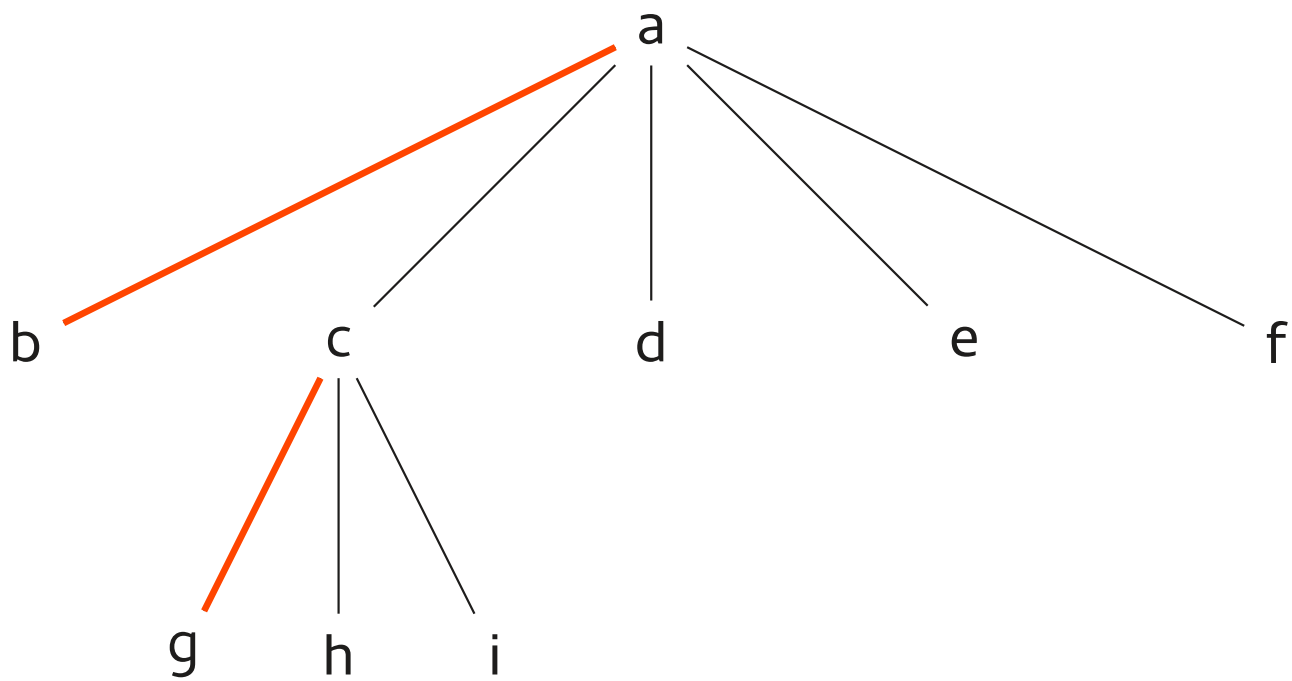
<title>Automates d'arbres (préliminaire)</title>

Encodage d'un arbre n-aire en arbre binaire :



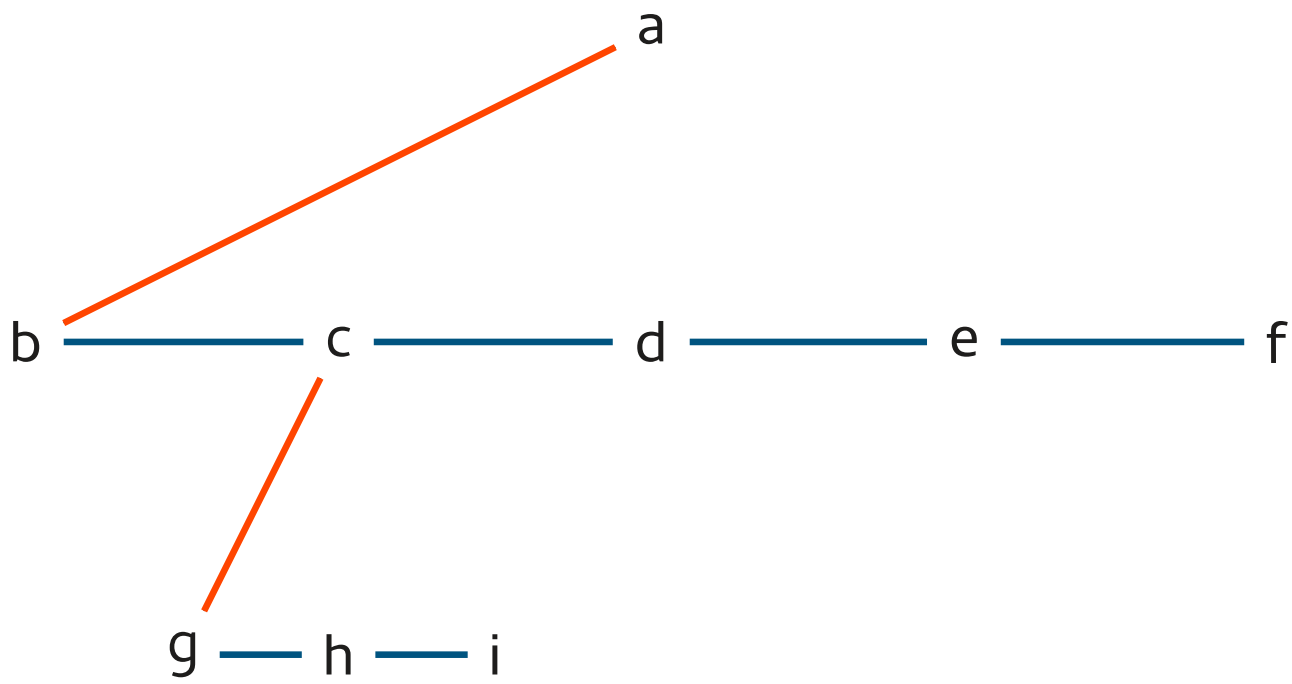
<title>Automates d'arbres (préliminaire)</title>

Encodage d'un arbre n-aire en arbre binaire :



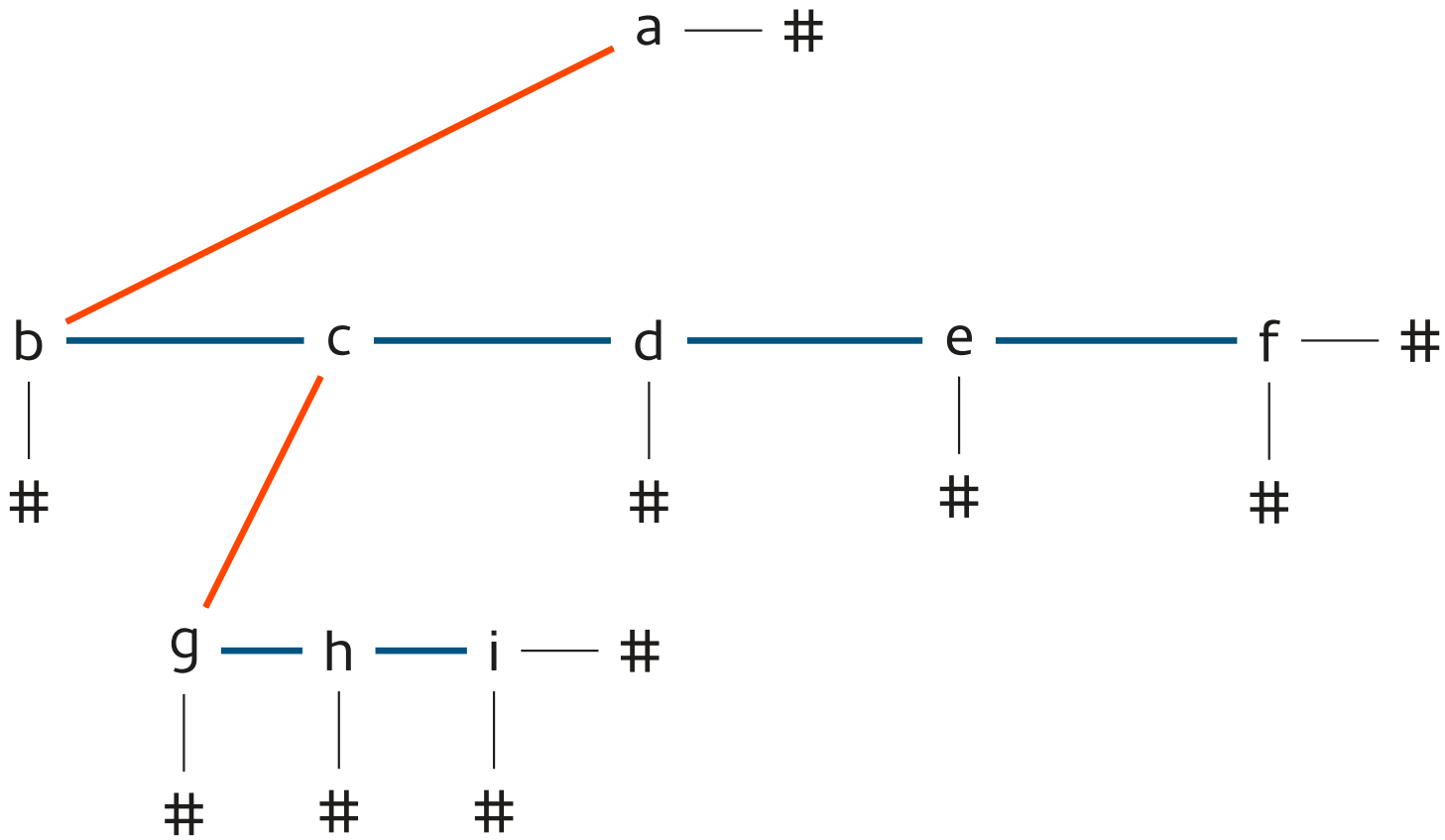
<title>Automates d'arbres (préliminaire)</title>

Encodage d'un arbre n-aire en arbre binaire :



<title>Automates d'arbres (préliminaire)</title>

Encodage d'un arbre n-aire en arbre binaire :



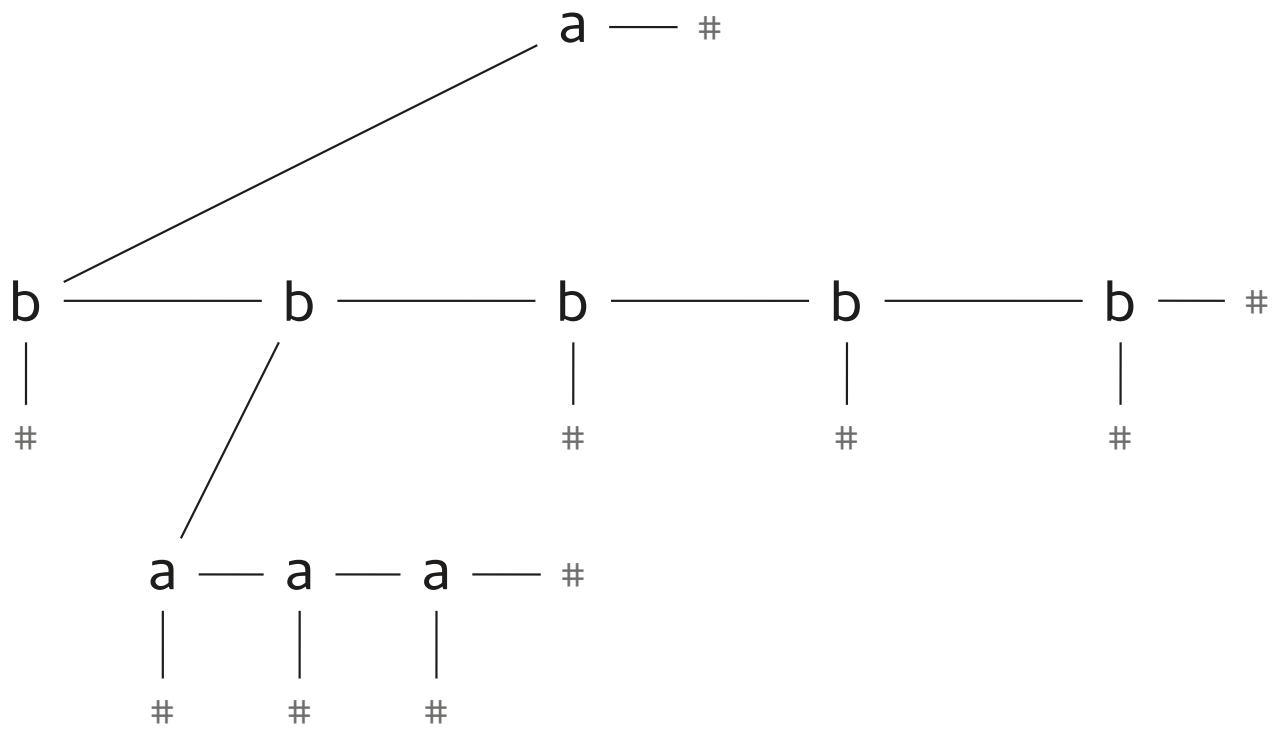
<title>Automates d'arbres</title>

Caractériser des ensembles d'arbres :
« Arbres ayant des a à profondeur paire et b à profondeur impaire (dans l'arbre n-aire)»

Definition (Automate d'arbres (*[Comon et al. 2007]*))
Un automate d'arbre est un tuple $(\Sigma, Q, T, B, \delta)$:

- Σ : alphabet (ensemble de symboles et leur arité)
- Q : ensemble d'états
- $T \subseteq Q$: états à la racine (Top)
- $B \subseteq Q$: états aux feuilles (Bottom)
- $\delta \subseteq Q \times \Sigma \times Q \times Q$: relation de transition

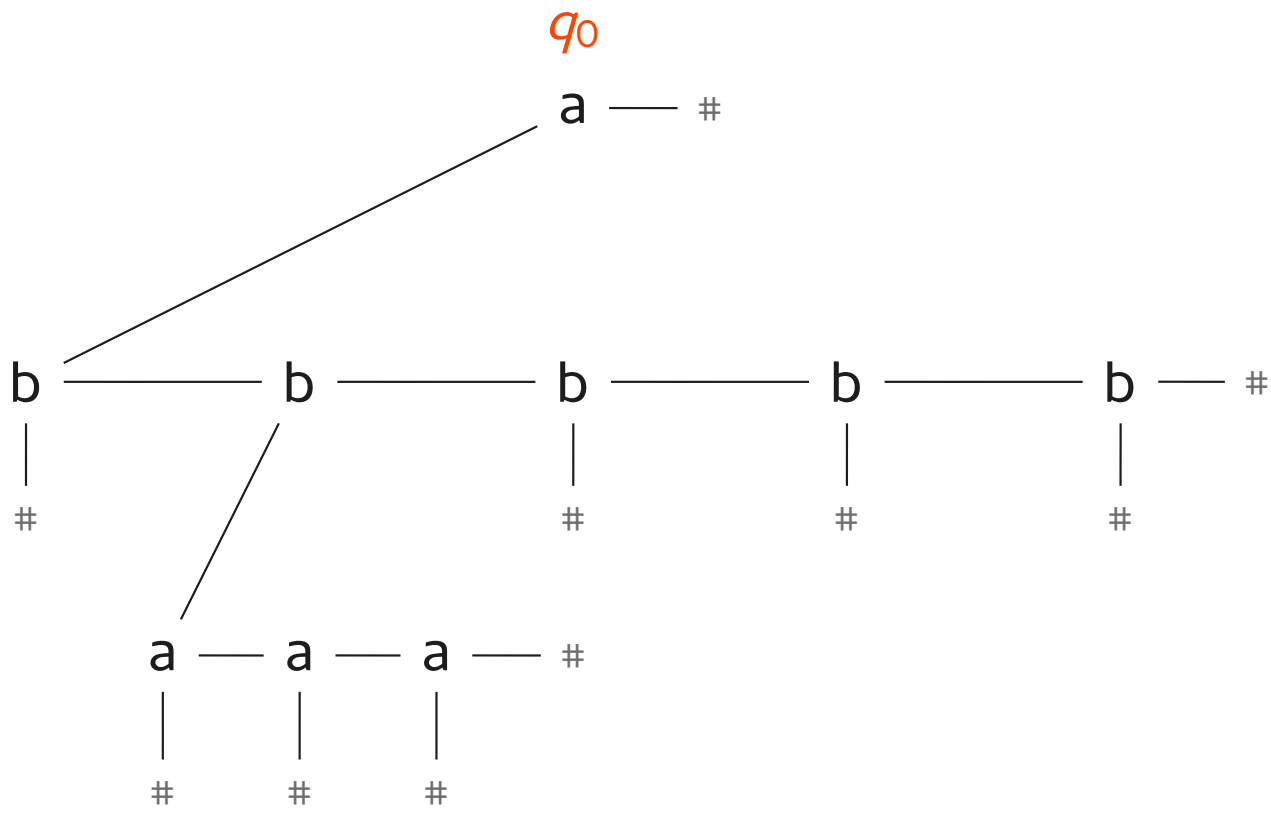
<title>Automate d'arbres </title>



$$A_1 = \{\mathsf{a}^2, \mathsf{b}^2, \#^0\}, \{q_0, q_1, q_\perp\}, \{q_0\}, \{q_0, q_1\},$$

q_0, a	(q_1, q_0)
q_0, b	$(q_\perp, q_\perp)$
q_1, b	(q_0, q_1)
q_1, a	$(q_\perp, q_\perp)$
$q_\perp, \_$	$(q_\perp, q_\perp)$

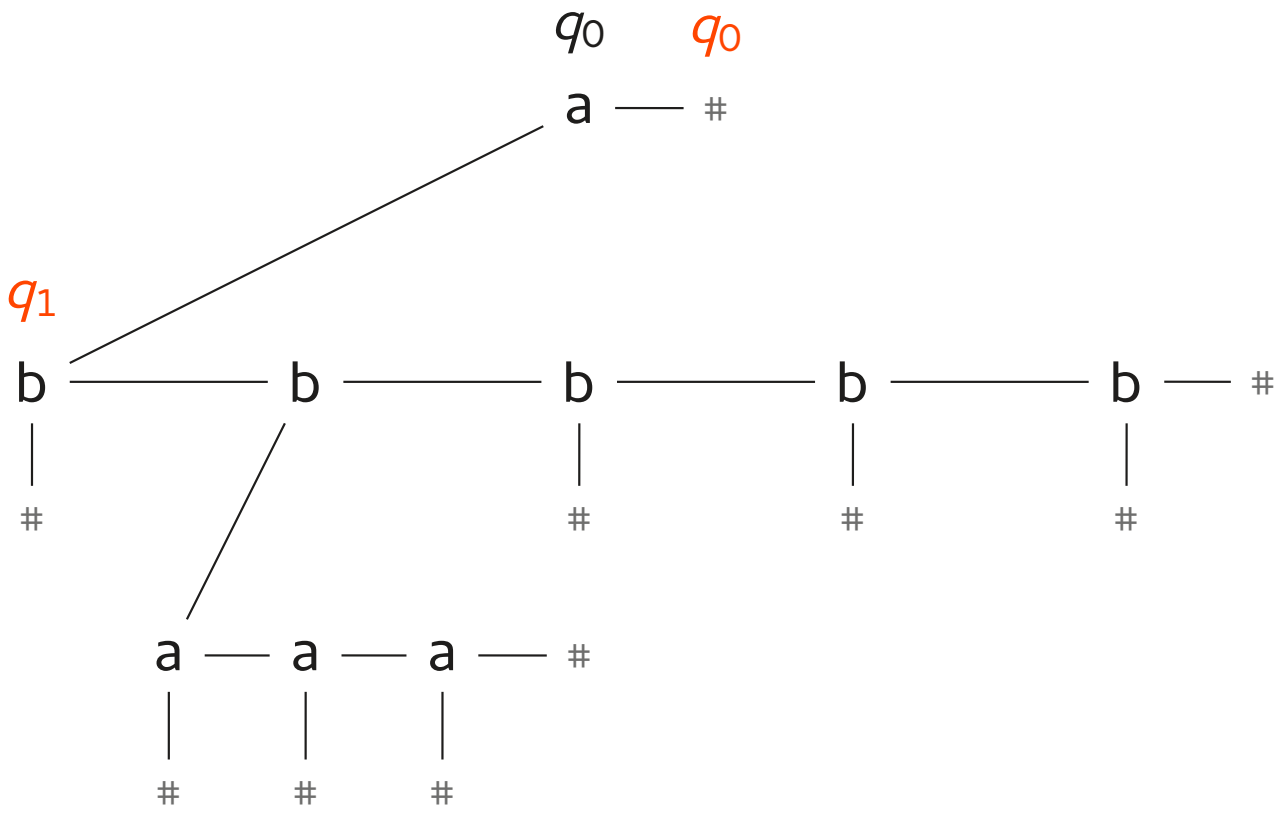
<title>Automate d'arbres </title>



$$A_1 = \{\mathsf{a}^2, \mathsf{b}^2, \#^0\}, \{q_0, q_1, q_\perp\}, \{q_0\}, \{q_0, q_1\},$$

q_0, a	(q_1, q_0)
q_0, b	$(q_\perp, q_\perp)$
q_1, b	(q_0, q_1)
q_1, a	$(q_\perp, q_\perp)$
$q_\perp, \_$	$(q_\perp, q_\perp)$

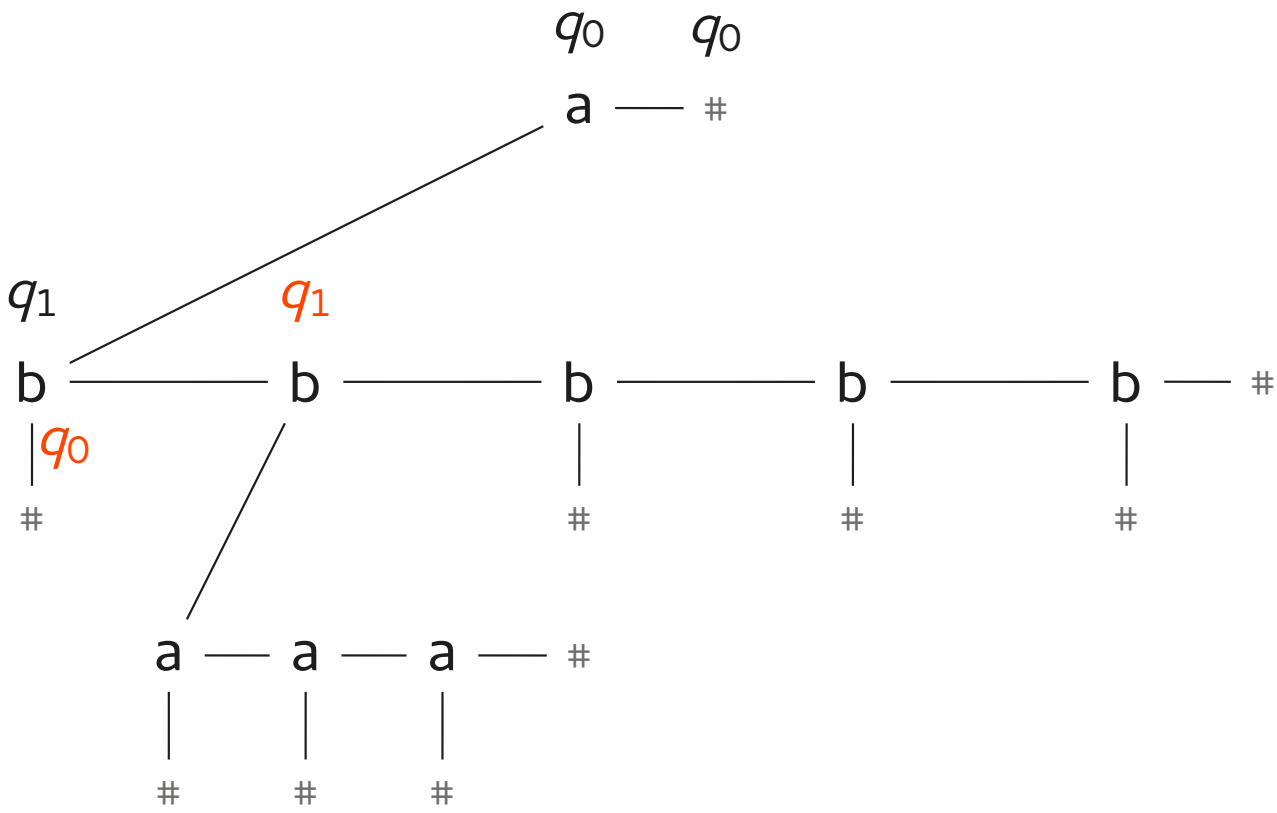
<title>Automate d'arbres </title>



$$A_1 = \{a^2, b^2, \#^0\}, \{q_0, q_1, q_\perp\}, \{q_0\}, \{q_0, q_1\},$$

q_0, a	(q_1, q_0)
q_0, b	$(q_\perp, q_\perp)$
q_1, b	(q_0, q_1)
q_1, a	$(q_\perp, q_\perp)$
$q_\perp, \_$	$(q_\perp, q_\perp)$

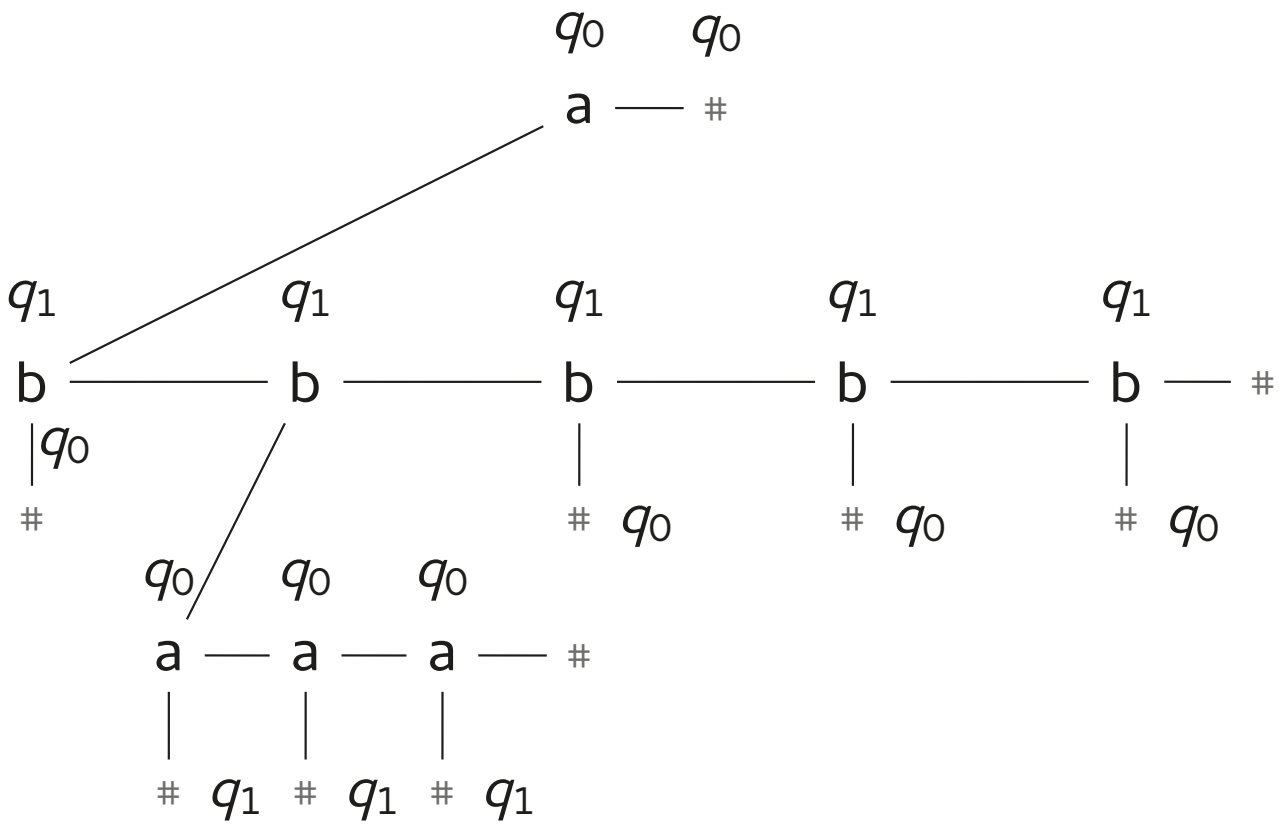
<title>Automate d'arbres </title>



$$A_1 = \{\mathsf{a}^2, \mathsf{b}^2, \#^0\}, \{q_0, q_1, q_\perp\}, \{q_0\}, \{q_0, q_1\},$$

q_0, a	(q_1, q_0)
q_0, b	$(q_\perp, q_\perp)$
q_1, b	(q_0, q_1)
q_1, a	$(q_\perp, q_\perp)$
$q_\perp, \_$	$(q_\perp, q_\perp)$

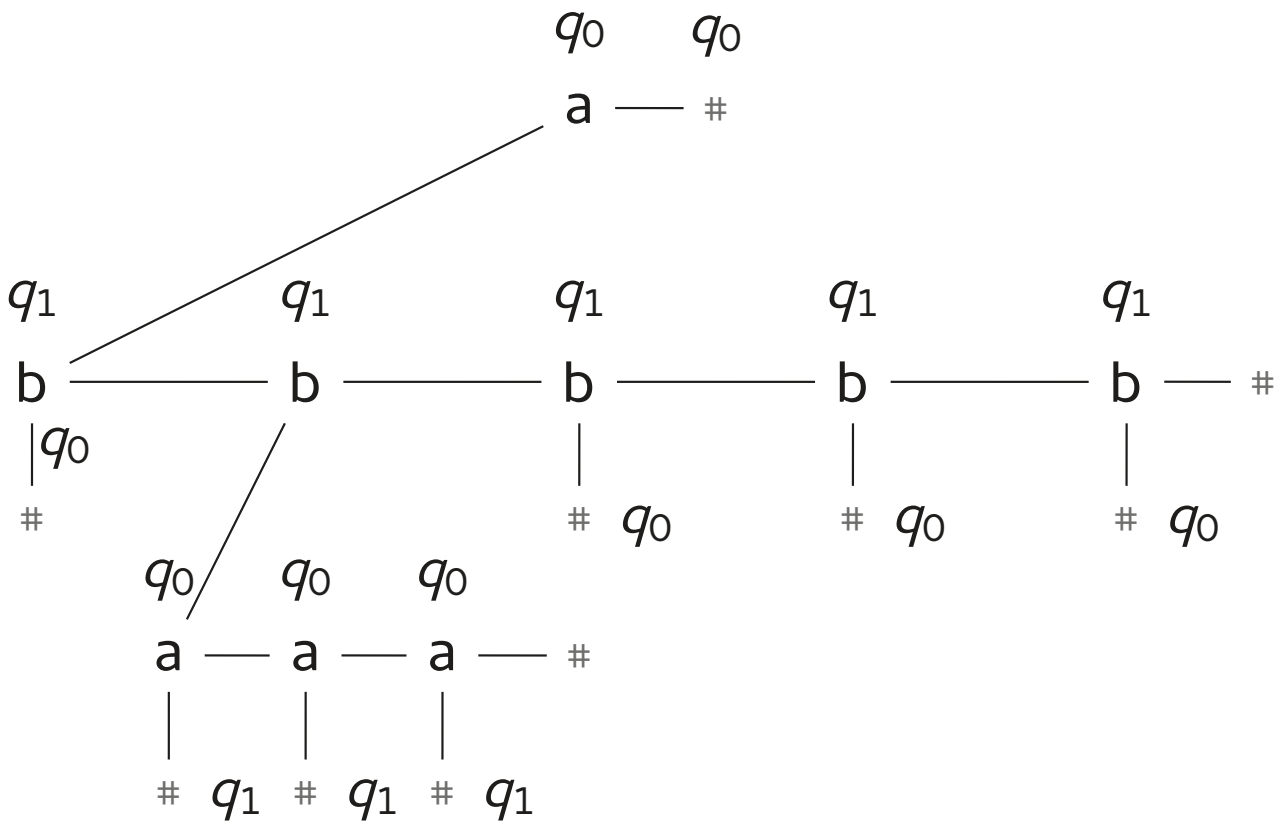
<title>Automate d'arbres </title>



$$A_1 = \{\mathsf{a}^2, \mathsf{b}^2, \#^0\}, \{q_0, q_1, q_\perp\}, \{q_0\}, \{q_0, q_1\},$$

q_0, a	(q_1, q_0)
q_0, b	$(q_\perp, q_\perp)$
q_1, b	(q_0, q_1)
q_1, a	$(q_\perp, q_\perp)$
$q_\perp, \_$	$(q_\perp, q_\perp)$

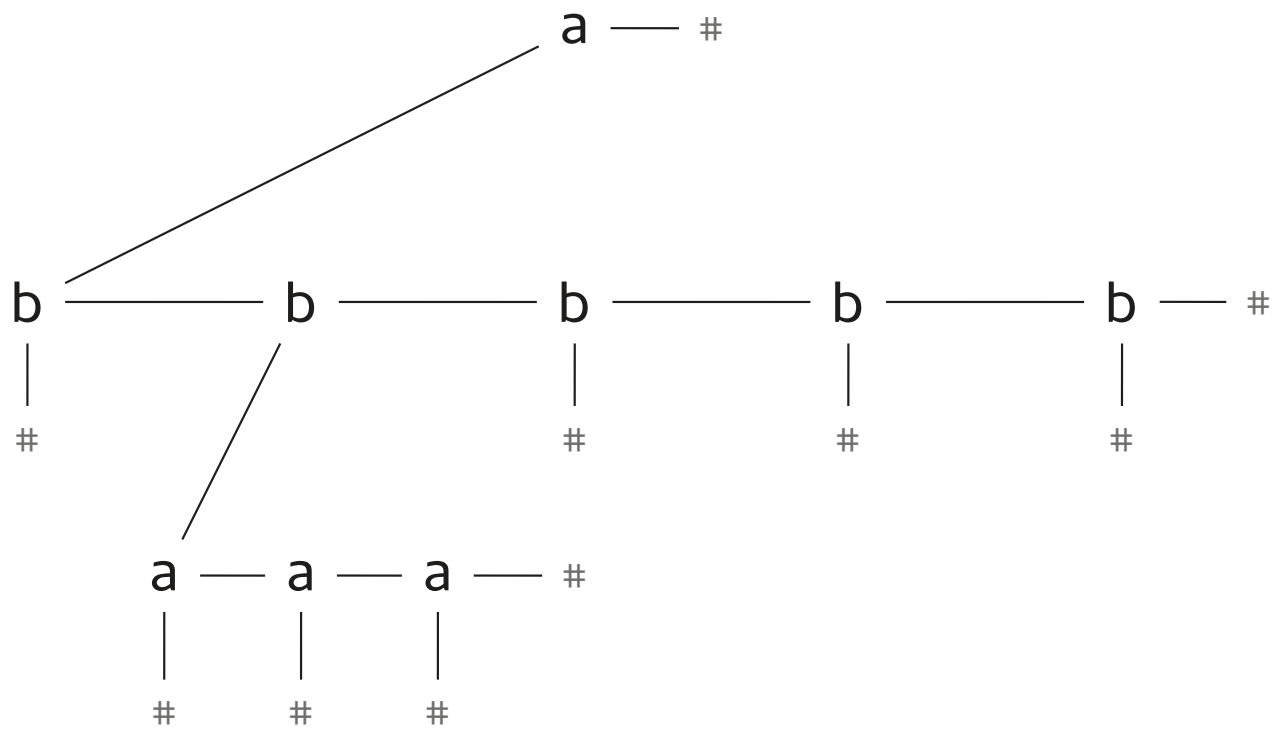
<title>Automate d'arbres descendant</title>



$$A_1 = \{\mathsf{a}^2, \mathsf{b}^2, \#^0\}, \{q_0, q_1, q_\perp\}, \{q_0\}, \{q_0, q_1\},$$

q_0, a	(q_1, q_0)
q_0, b	$(q_\perp, q_\perp)$
q_1, b	(q_0, q_1)
q_1, a	$(q_\perp, q_\perp)$
$q_\perp, \_$	$(q_\perp, q_\perp)$

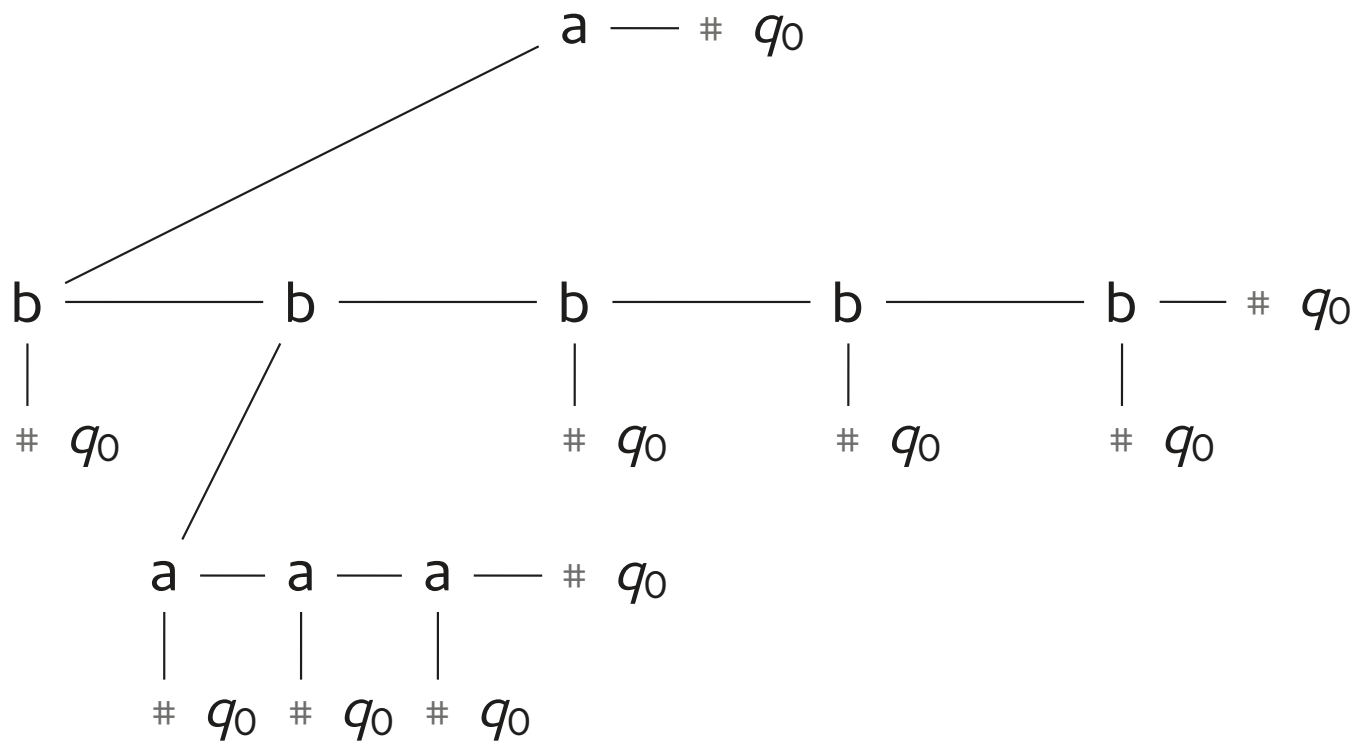
<title>Automate d'arbres </title>



$$A_2 = \{a^2, b^2, \#^0\}, \{q_0, q_1, q_2, q_\perp\}, \{q_1\}, \{q_0\},$$

$$\begin{array}{ll} (q_0|q_2, q_0|q_1, a) & q_1 \\ (q_0|q_1, q_0|q_2, b) & q_2 \\ \ldots & q_\perp \end{array}$$

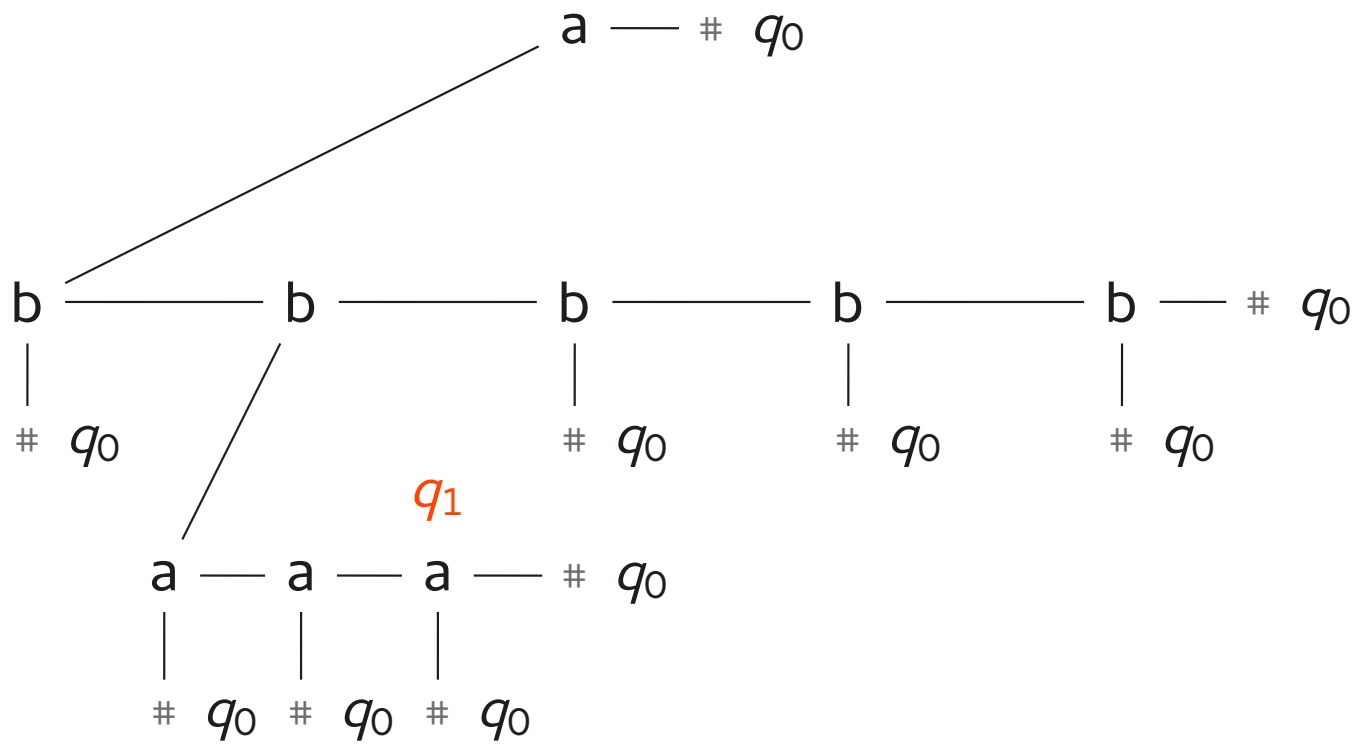
<title>Automate d'arbres </title>



$$A_2 = \{a^2, b^2, \#^0\}, \{q_0, q_1, q_2, q_\perp\}, \{q_1\}, \{q_0\},$$

$$\begin{array}{ll} (q_0|q_2, q_0|q_1, a) & q_1 \\ (q_0|q_1, q_0|q_2, b) & q_2 \\ \ldots & q_\perp \end{array}$$

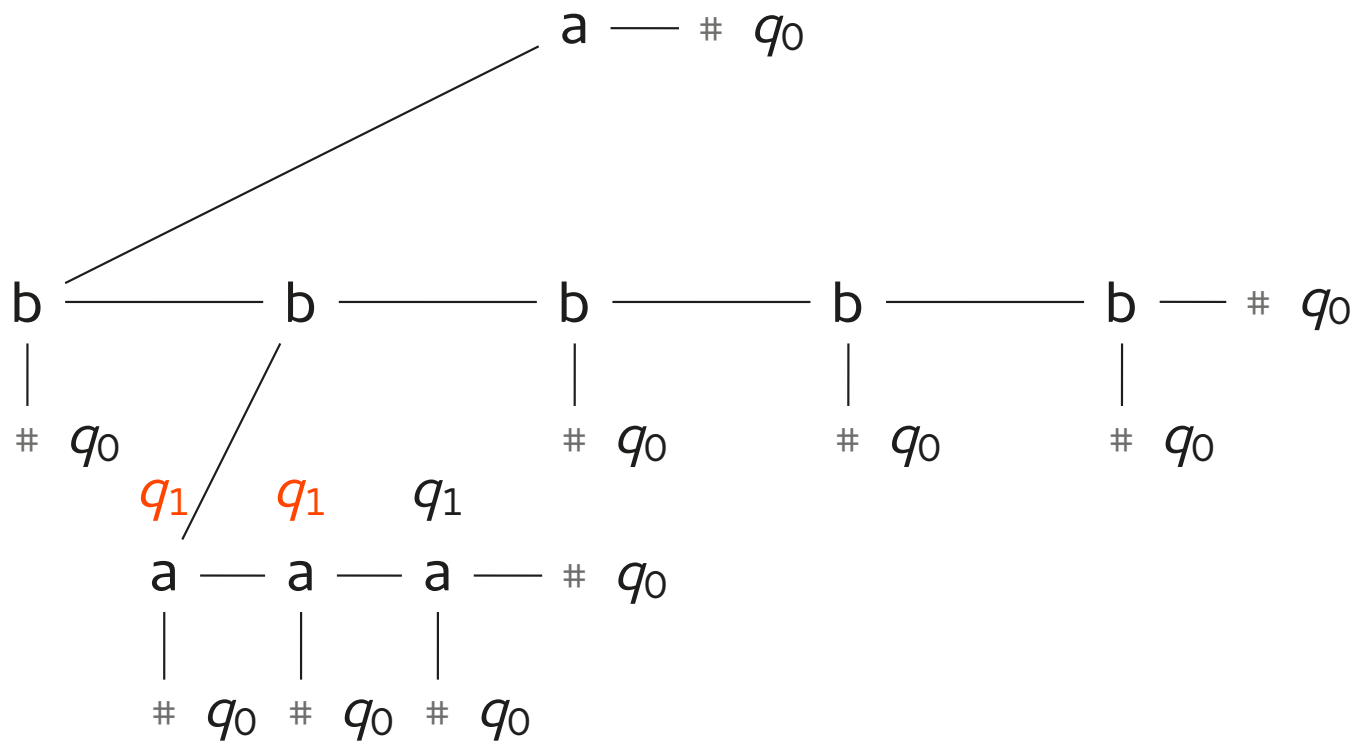
<title>Automate d'arbres </title>



$$A_2 = \{\mathsf{a}^2, \mathsf{b}^2, \#^0\}, \{q_0, q_1, q_2, q_\perp\}, \{q_1\}, \{q_0\},$$

$$\begin{array}{ll} (q_0|q_2, q_0|q_1, \mathsf{a}) & q_1 \\ (q_0|q_1, q_0|q_2, \mathsf{b}) & q_2 \\ \ldots & q_\perp \end{array}$$

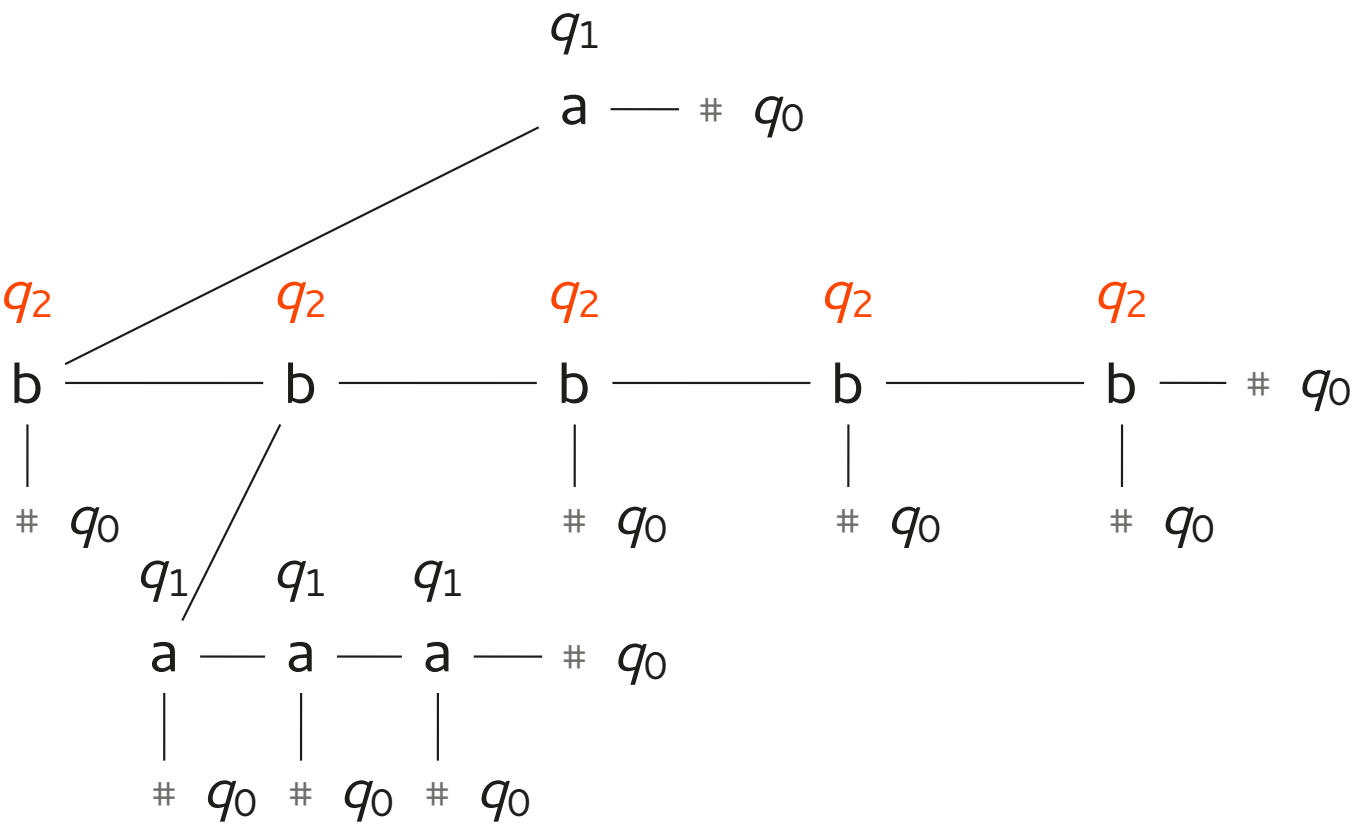
<title>Automate d'arbres </title>



$$A_2 = \{\mathsf{a}^2, \mathsf{b}^2, \#^0\}, \{q_0, q_1, q_2, q_\perp\}, \{q_1\}, \{q_0\},$$

$$\begin{array}{ll} (q_0|q_2, q_0|q_1, \mathsf{a}) & q_1 \\ (q_0|q_1, q_0|q_2, \mathsf{b}) & q_2 \\ \ldots & q_\perp \end{array}$$

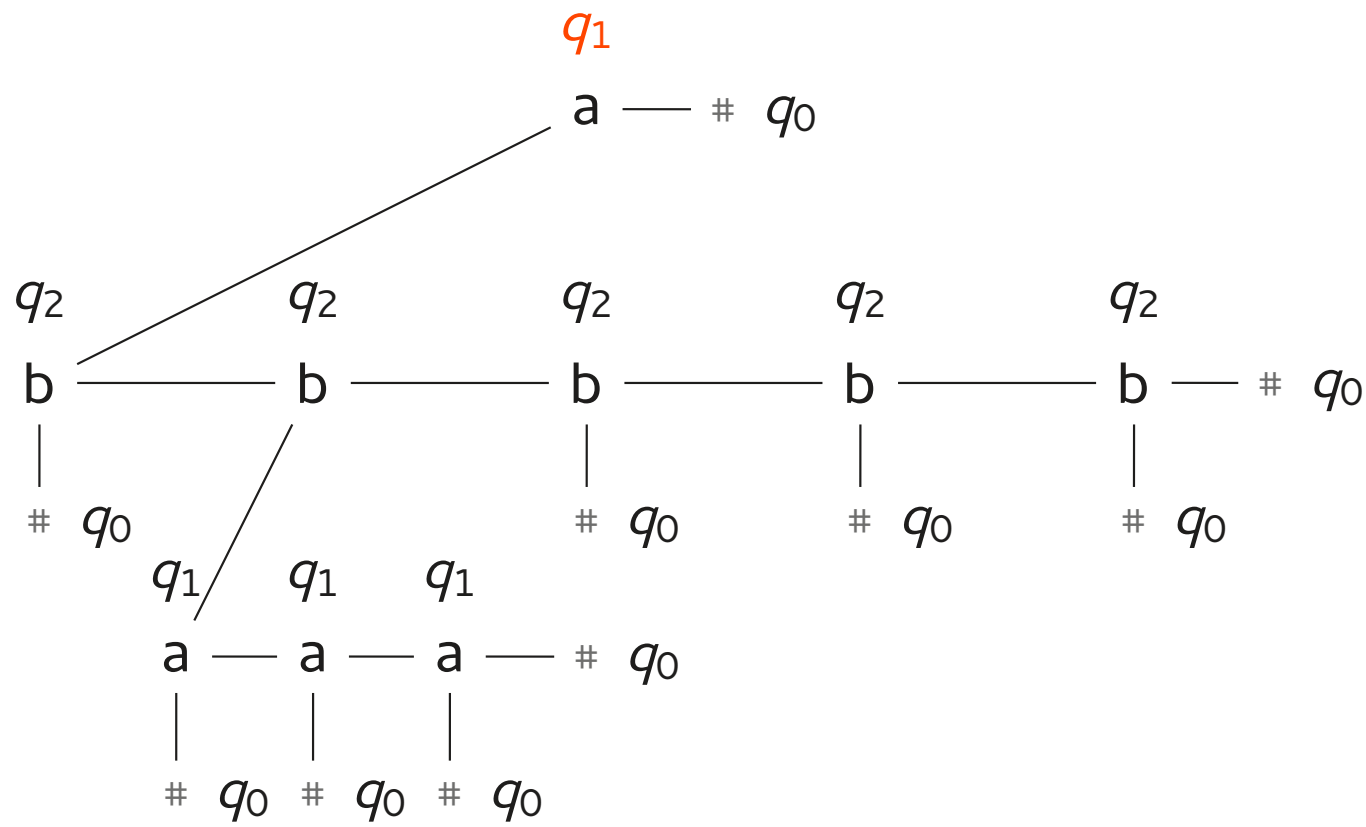
<title>Automate d'arbres </title>



$$A_2 = \{\mathsf{a}^2, \mathsf{b}^2, \#^0\}, \{q_0, q_1, q_2, q_\perp\}, \{q_1\}, \{q_0\},$$

$$\begin{array}{ll} (q_0|q_2, q_0|q_1, \mathsf{a}) & q_1 \\ (q_0|q_1, q_0|q_2, \mathsf{b}) & q_2 \\ \ldots & q_\perp \end{array}$$

<title>Automate d'arbres ascendant</title>



$$A_2 = \{\mathsf{a}^2, \mathsf{b}^2, \#^0\}, \{q_0, q_1, q_2, q_\perp\}, \{q_1\}, \{q_0\},$$

$$\begin{array}{ll} (q_0|q_2, q_0|q_1, \mathsf{a}) & q_1 \\ (q_0|q_1, q_0|q_2, \mathsf{b}) & q_2 \\ \ldots & q_\perp \end{array}$$

<title>Automates d'arbres</title>

- déterministes (D) / non-déterministes (N)
- top-down (TD) / bottom-up (BU)

<title>Automates d'arbres</title>

- déterministes (D) / non-déterministes (N)
- top-down (TD) / bottom-up (BU)

Quelques propriétés :

1. D.TD. $\subsetneq$ D.BU. $=$ N.TD. $=$ N.BU. $=$ Langages Réguliers d'arbres
   Exercice : trouver un contre exemple pour D.TD. $\subsetneq$ D.BU.
2. N.BU. : opérations d'union, intersection, complémentaire
3. N.BU. peuvent être déterminisé (explosion du nombre d'états)
4. Appartenance :

   déterministes $O(|T|)$, en *streaming* pour les D.TD.
   non-det. $O(|\delta| \times |T|)$ ou $O(2^{|Q|} + |T|)$
5. Inclusion, vide, équivalence :

   déterministes décidable en temps polynomial
   non-det. EXPTIME-complet

<title>Digression récursive</title>

```
(* Arbres binaires *)
type t = Nil | Node of string * t * t

(* Appartenance pour D. TD. *)
let rec td_mem t q =
    match t with
     Nil ->  q ∈ B
   | Node (l, t1, t2) ->
          let q1, q2 = δ(q,l) in
          if td_mem t1 q1 then td_mem t2 q2
          else false
```

<title>Digression récursive</title>

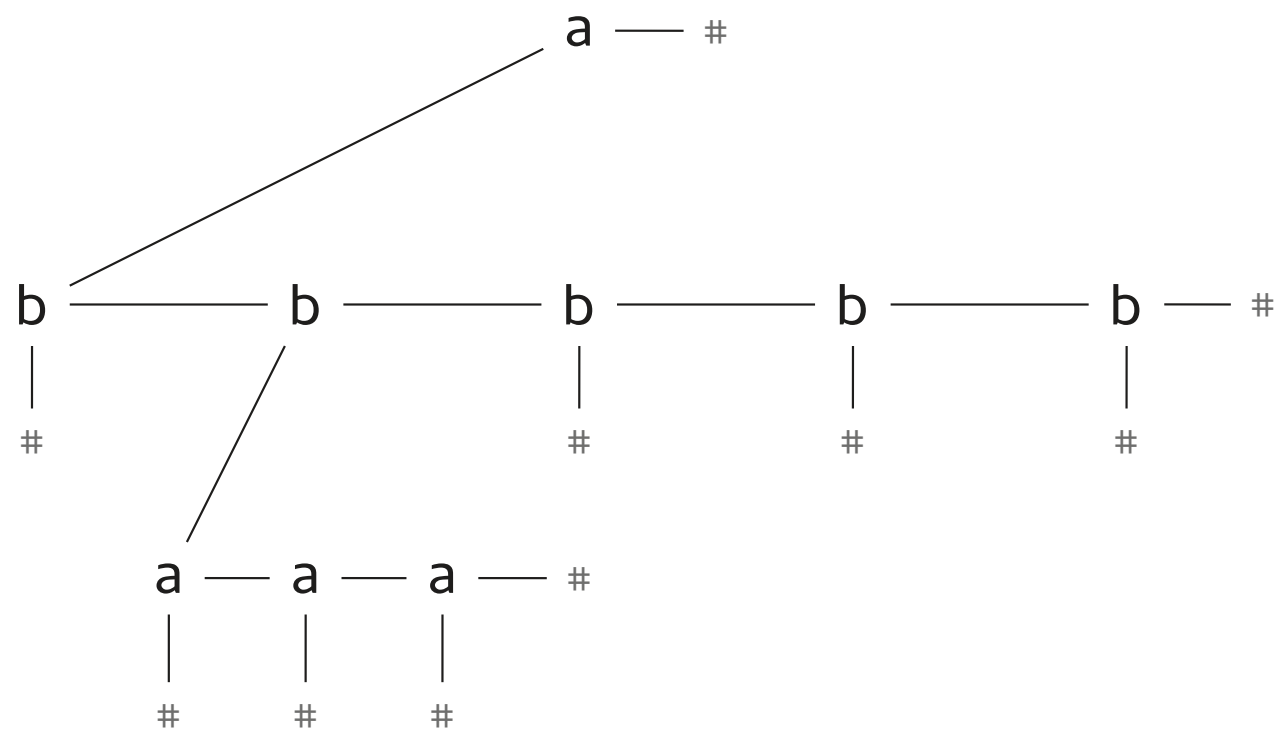
```
(* Arbres binaires *)
type t = Nil | Node of string * t * t

(* Appartenance pour D. TD. *)
let rec td_mem t q =
    match t with
     Nil ->  ...
   | Node (l, t1, t2) ->
          ...
          if td_mem t1 q1 then td_mem t2 q2
          else false
```

Les fonctions qui ont ce schéma sont *streamables*

<title>Automate avec marquage</title>

On rajoute un ensemble d'états de selection

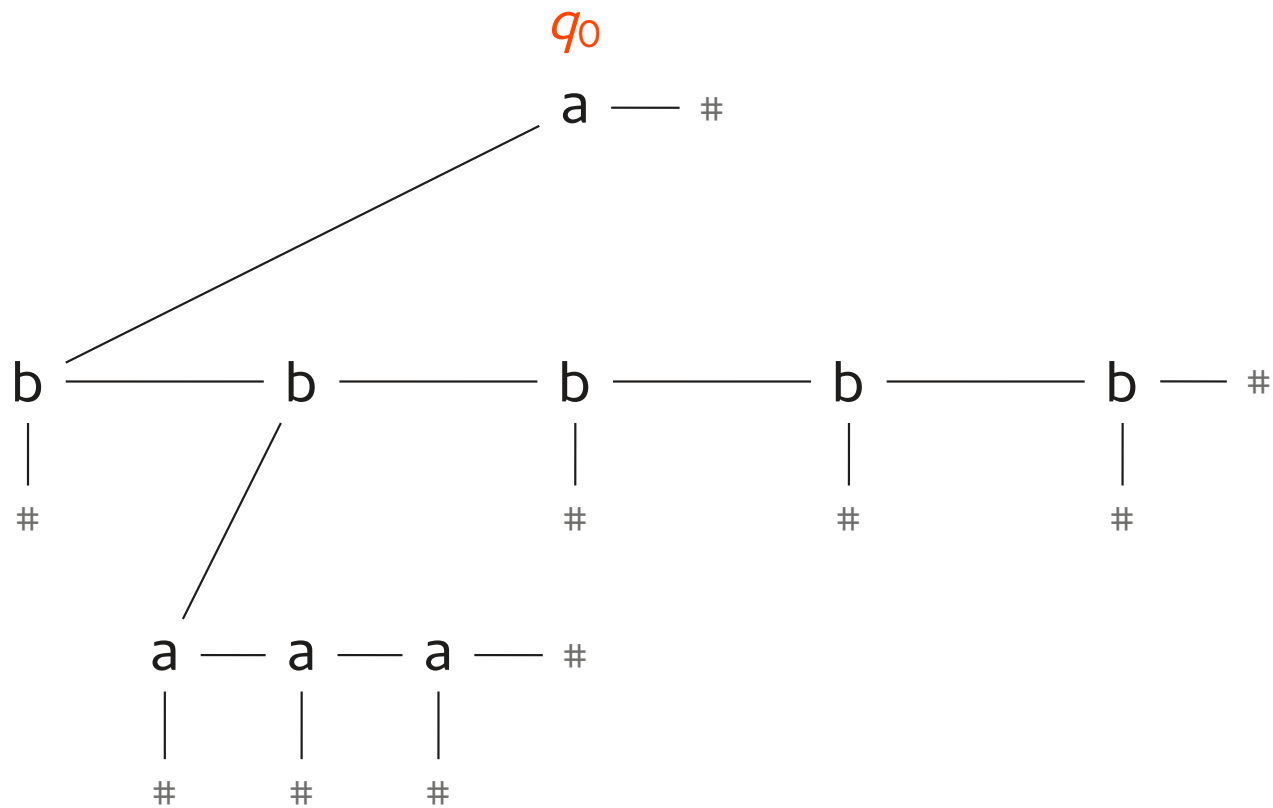


$$A_1 = \{a^2, b^2, \#^0\}, \{q_0, q_1, q_\perp\}, \{q_0\}, \{q_0, q_1\}, \{\mathbf{q_1}\},$$

q_0, a	(q_1, q_0)
q_0, b	$(q_\perp, q_\perp)$
q_1, b	(q_0, q_1)
q_1, a	$(q_\perp, q_\perp)$
$q_\perp, \_$	$(q_\perp, q_\perp)$

<title>Automate avec marquage</title>

On rajoute un ensemble d'états de selection

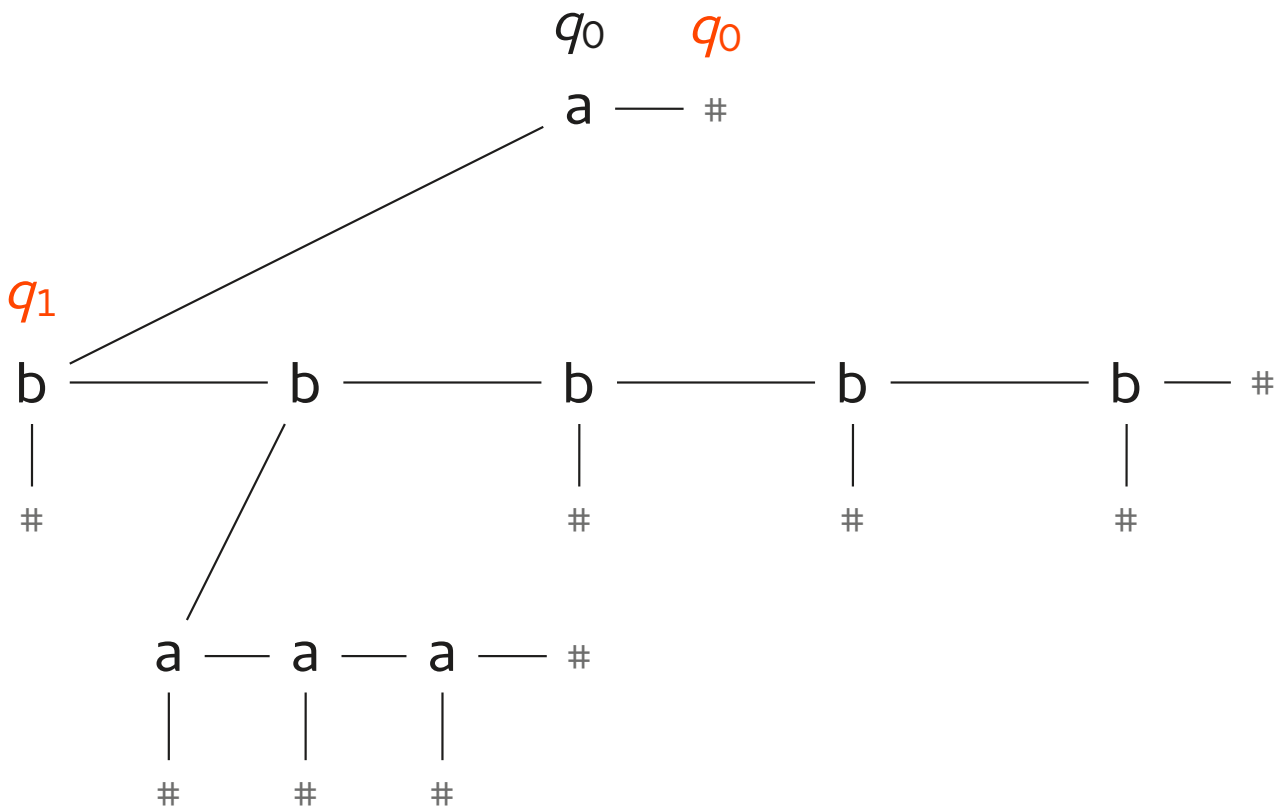


$$A_1 = \{a^2, b^2, \#^0\}, \{q_0, q_1, q_\perp\}, \{q_0\}, \{q_0, q_1\}, \{\mathbf{q_1}\},$$

q_0, a	(q_1, q_0)
q_0, b	$(q_\perp, q_\perp)$
q_1, b	(q_0, q_1)
q_1, a	$(q_\perp, q_\perp)$
$q_\perp, \_$	$(q_\perp, q_\perp)$

<title>Automate avec marquage</title>

On rajoute un ensemble d'états de selection

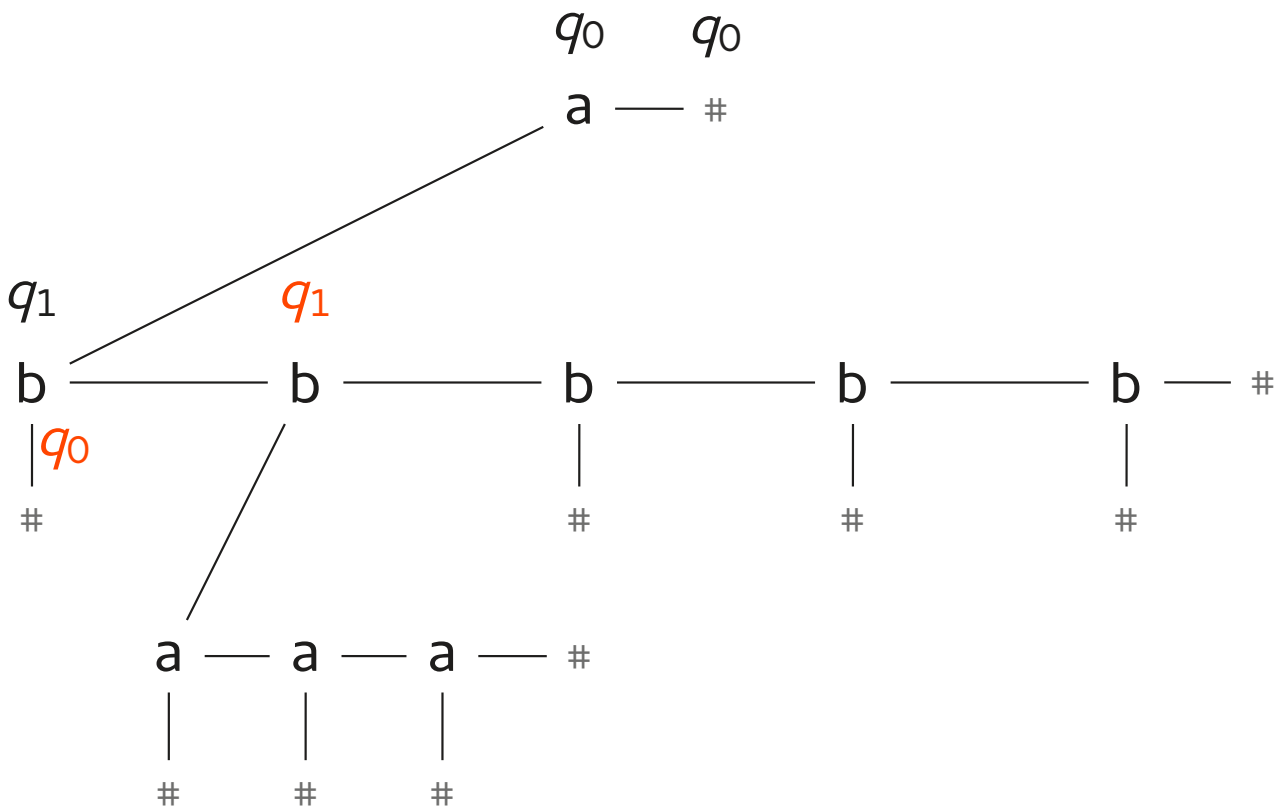


$$A_1 = \{a^2, b^2, \#^0\}, \{q_0, q_1, q_\perp\}, \{q_0\}, \{q_0, q_1\}, \{\mathbf{q_1}\},$$

q_0, a	(q_1, q_0)
q_0, b	$(q_\perp, q_\perp)$
q_1, b	(q_0, q_1)
q_1, a	$(q_\perp, q_\perp)$
$q_\perp, \_$	$(q_\perp, q_\perp)$

<title>Automate avec marquage</title>

On rajoute un ensemble d'états de selection

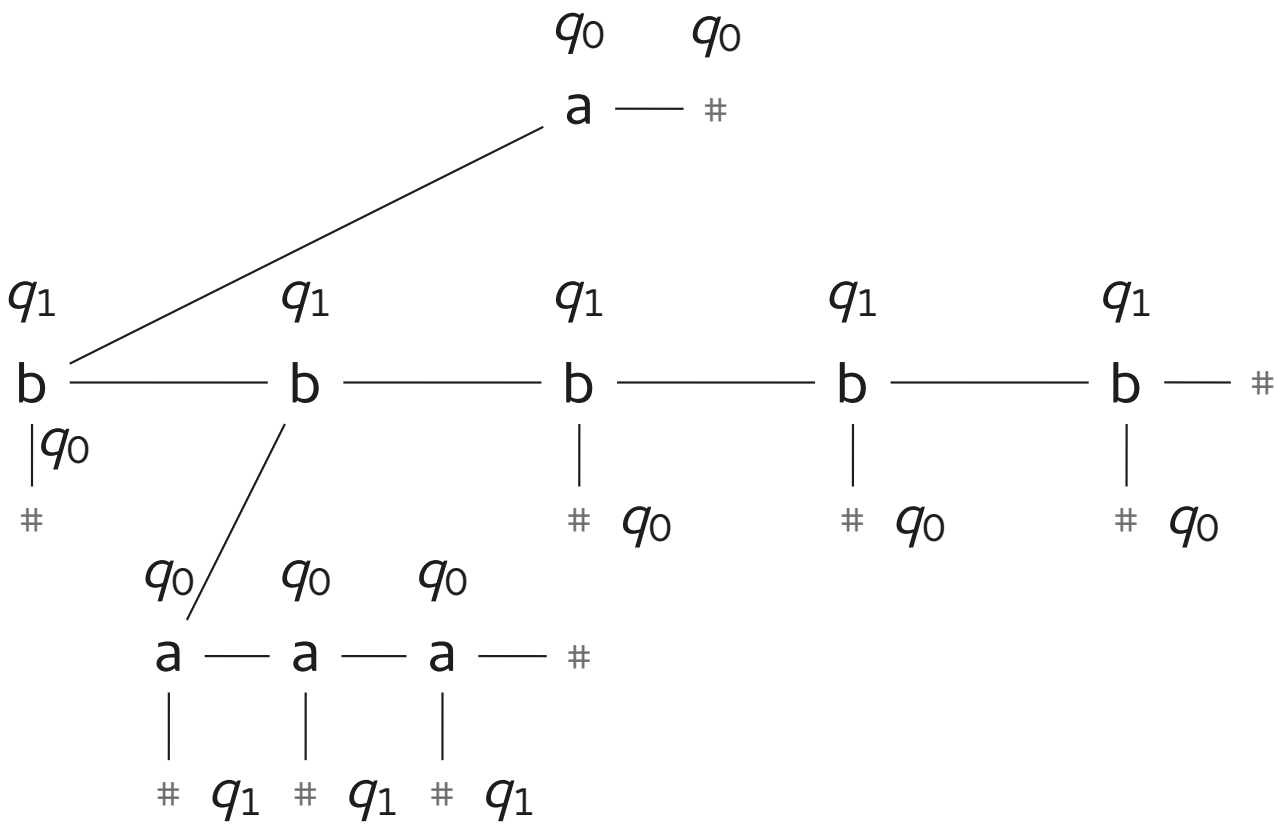


$$A_1 = \{a^2, b^2, \#^0\}, \{q_0, q_1, q_\perp\}, \{q_0\}, \{q_0, q_1\}, \{\mathbf{q_1}\},$$

q_0, a	(q_1, q_0)
q_0, b	$(q_\perp, q_\perp)$
q_1, b	(q_0, q_1)
q_1, a	$(q_\perp, q_\perp)$
$q_\perp, \_$	$(q_\perp, q_\perp)$

<title>Automate avec marquage</title>

On rajoute un ensemble d'états de selection



$A_1 = \{a^2, b^2, \#^0\}, \{q_0, q_1, q_\perp\}, \{q_0\}, \{q_0, q_1\}, \{\mathbf{q_1}\},$

q_0, a	(q_1, q_0)
q_0, b	$(q_\perp, q_\perp)$
q_1, b	(q_0, q_1)
q_1, a	$(q_\perp, q_\perp)$
$q_\perp, \_$	$(q_\perp, q_\perp)$

<title>Automate avec marquage</title>

- Bien définis dans le cas déterministe
- Le cas non déterministe est plus exotique
  - Plusieurs parcours acceptants
  - Nœuds marqués lors d'un parcours accpetant ? lors de tous ?

<title>Automate avec marquage</title>

- Bien définis dans le cas déterministe
- Le cas non déterministe est plus exotique
  - Plusieurs parcours acceptants
  - Nœuds marqués lors d'un parcours accpetant ? lors de tous ?

Digression Unixienne :

- Perl utilise des automates *déterministes* pour les expression régulières
- Complexité en $O(2^{|R|} + |D|)$ au lieu de $O(|R| \times |D|)$!
- Lié aux *back-references* `\1`, `\2`, ...
- Idem pour `lex`, `awk`, `grep`, ...

3. ℂDuce et le sous-typage sémantique

<title>ℂDuce (http ://www.cduce.org)</title>

[Frisch et al. 2008]

- Langage fonctionnel, d'ordre supérieur
- Adapté à la manipulation de documents XML
- Surcharge/polymorphisme ad-hac
- *Pattern-matching* efficace

<title>ℂDuce : un exemple</title>

```
include "xhtml.cd"
type t_phone = <phone>[ ('0'--'9')+ ]
type t_email = <email>[ (Char\ '@')+ '@' (Char\'@')+ ]
type t_name = <name>[ <first>[Char+] <last>[Char+] ]
type t_contact = <contact>[ t_name t_email* t_phone* ]
type t_addressbook = <addressbook>[ t_contact* ]

let to_li (x : t_email|t_phone) : t_li =
      match x with
        <email>s -> <li>("couriel: " @ s)
      | <_ >(s & [ '0' ('6' | '7') _* ])  -> <li>("portable :" @ s)
      | <_ >s  -> <li>("téléphone: " @ s)
```

<title>ℂDuce : un exemple</title>

```
include "xhtml.cd"
type t_phone = <phone>[ ('0'--'9')+ ]
type t_email = <email>[ (Char\ '@')+ '@' (Char\'@')+ ]
type t_name = <name>[ <first>[Char+] <last>[Char+] ]
type t_contact = <contact>[ t_name t_email* t_phone* ]
type t_addressbook = <addressbook>[ t_contact* ]

let to_li (x : t_email|t_phone) : t_li =
      match x with
        <email>s -> <li>("couriel: " @ s)
      | <_ >(s & [ '0' ('6' | '7') _* ])  -> <li>("portable :" @ s)
      | <_ >s  -> <li>("téléphone: " @ s)
```

- Constructeurs

<title>ℂDuce : un exemple</title>

```
include "xhtml.cd"
type t_phone = <phone>[ ('0'--'9')+ ]
type t_email = <email>[ (Char\ '@')+ '@' (Char\'@')+ ]
type t_name = <name>[ <first>[Char+] <last>[Char+] ]
type t_contact = <contact>[ t_name t_email* t_phone* ]
type t_addressbook = <addressbook>[ t_contact* ]

let to_li (x : t_email|t_phone) : t_li =
      match x with
        <email>s -> <li>("couriel: " @ s)
      | <_ >(s & [ '0' ('6' | '7') _* ])  -> <li>("portable :" @ s)
      | <_ >s  -> <li>("téléphone: " @ s)
```

- Constructeurs
- Expressions régulières de types

<title>ℂDuce : un exemple</title>

```
include "xhtml.cd"
type t_phone = <phone>[ ('0'--'9')+ ]
type t_email = <email>[ (Char\ '@')+ '@' (Char\'@')+ ]
type t_name = <name>[ <first>[Char+] <last>[Char+] ]
type t_contact = <contact>[ t_name t_email* t_phone* ]
type t_addressbook = <addressbook>[ t_contact* ]

let to_li (x : t_email|t_phone) : t_li =
      match x with
        <email>s -> <li>("couriel: " @ s)
      | <_ >(s & [ '0' ('6' | '7') _* ])  -> <li>("portable :" @ s)
      | <_ >s  -> <li>("téléphone: " @ s)
```

- Constructeurs
- Expressions régulières de types
- Union, intersection, singleton, différence

<title>ℂDuce : algèbre de type</title>

$$
\begin{array}{rll}
t ::= & \texttt{int} \mid \texttt{string} \mid \ldots & \text{(types de base)} \\
\mid & \texttt{`nil} \mid 42 \mid \ldots & \text{(types singletons)} \\
\mid & (t,t) & \text{(produit)} \\
\mid & t \to t & \text{(flêche)} \\
\mid & t \mid t & \text{(union)} \\
\mid & t \,\&\, t & \text{(intersection)} \\
\mid & \neg t & \text{(négation)} \\
\mid & \texttt{empty} & \text{(bottom)} \\
\mid & \texttt{any} & \text{(top)} \\
\mid & \mu T.t & \text{(types récursifs)} \\
\mid & T & \text{(variable de récursion)}
\end{array}
$$

Expressions régulières :

$$(\texttt{int}, \mu T.\texttt{`nil} \mid (\texttt{int}, T)) \equiv \texttt{[Int+]}$$

<title>Problème de l'approche syntaxique</title>

$$\frac{\Gamma \vdash e_1 : t \to s \quad \Gamma \vdash e_2 : u \quad u \leq t}{\Gamma \vdash e_1\, e_2 : s}$$

Difficile de définir $u \leq t$ syntaxiquement :

- On veut prendre en compte les propriétés de distributivité, commutativité de **&** et |
- Beaucoup de cas à gérer
- Difficile d'avoir un algorithme syntaxique complet

<title>Sous-typage sémantique</title>

Definition (Sous-typage sémantique)

$$s \leq t \Leftrightarrow [\![s]\!] \subseteq [\![t]\!]$$

$[\![\_]\!]$: interprétation ensembliste des types
Équivalence **sémantique** des types :

- $(\texttt{int},\texttt{int})|(2,4) \equiv (\texttt{int},\texttt{int})$
- $\texttt{[intint* bool? bool+]} \equiv \texttt{[int+ bool+]}$
- $(t_1|t_2)\&(t_1|t_3) \equiv t_1|t_2\&t_3$

Il faut pouvoir décider $[\![s]\!] \subseteq [\![t]\!] \Leftrightarrow [\![s]\!] \cap \overline{[\![t]\!]} = \varnothing$.

<title>Implantation par automates d'arbres</title>

$$[\ \texttt{int*}\ ] \equiv \mu T.\texttt{`nil} \mid (\texttt{int}, T)$$

<title>Implantation par automates d'arbres</title>

$$[\ \texttt{int*}\ ] \equiv \mu T.\texttt{`nil} \mid (\texttt{int}, T)$$

$$T \equiv \texttt{`nil} \mid (\texttt{int}, T)$$

<title>Implantation par automates d'arbres</title>

$$[\ \texttt{int*}\ ] \equiv \mu T.\texttt{`nil} \mid (\texttt{int}, T)$$

$$T \equiv \texttt{`nil} \mid (\texttt{int}, T)$$

$$\begin{aligned} T &\equiv \texttt{`nil} \\ T &\equiv (\texttt{int}, T) \end{aligned}$$

<title>Implantation par automates d'arbres</title>

$$[\ \texttt{int*}\ ] \equiv \mu T.\texttt{`nil} \mid (\texttt{int}, T)$$

$$T \equiv \texttt{`nil} \mid (\texttt{int}, T)$$

$$\begin{aligned} T &\equiv \texttt{`nil} \\ T &\equiv (\texttt{int}, T) \end{aligned}$$

$$\begin{array}{ll} q_T, \texttt{`nil} & q_\#, q_\# \\ q_T, \times & q_{\texttt{int}}, q_T \end{array}$$

<title>Implantation par automates d'arbres</title>

$$[\ \texttt{int*}\ ] \equiv \mu T.\texttt{`nil} \mid (\texttt{int}, T)$$

$$T \equiv \texttt{`nil} \mid (\texttt{int}, T)$$

$$\begin{aligned} T &\equiv \texttt{`nil} \\ T &\equiv (\texttt{int}, T) \end{aligned}$$

$$\begin{array}{ll} q_T, \texttt{`nil} & q_\#, q_\# \\ q_T, \times & q_{\texttt{int}}, q_T \end{array}$$

Types récursifs $\equiv$ Automates d'arbres !
Union, intersection, complément, test du vide,...

<title>Implantation par automates d'arbres</title>

On souhaite décider $[\![t]\!] \subseteq \varnothing$. Intuition :

- On met t en forme normale disjonctive :

$$t \equiv \left(\bigvee_{\mathtt{any}_{\mathtt{basic}}} \bigwedge t_i \wedge \bigwedge \neg t_j \right) \vee \left(\bigvee_{\mathtt{any}_{\times}} \bigwedge t_i \wedge \bigwedge \neg t_j \right) \vee \cdots$$

- On utilise des automates d'arbres pour décider le vide des types récursifs
- Représentation ad-hoc pour chaque type de base (ex : les entiers sont représentés comme des ensemble d'intervalles maximaux)

<title>Implantation des motifs</title>

```
match v with
<foo>[ x&t_bar _* y ] -> ...
```

<title>Implantation des motifs</title>

```
match v with
<foo>[ x&t_bar _* y ] -> ...
```

- Syntaxiquement, les motifs sont des types avec des variables de capture

<title>Implantation des motifs</title>

```
match v with
<foo>[ x&t_bar _* y ] -> ...
```

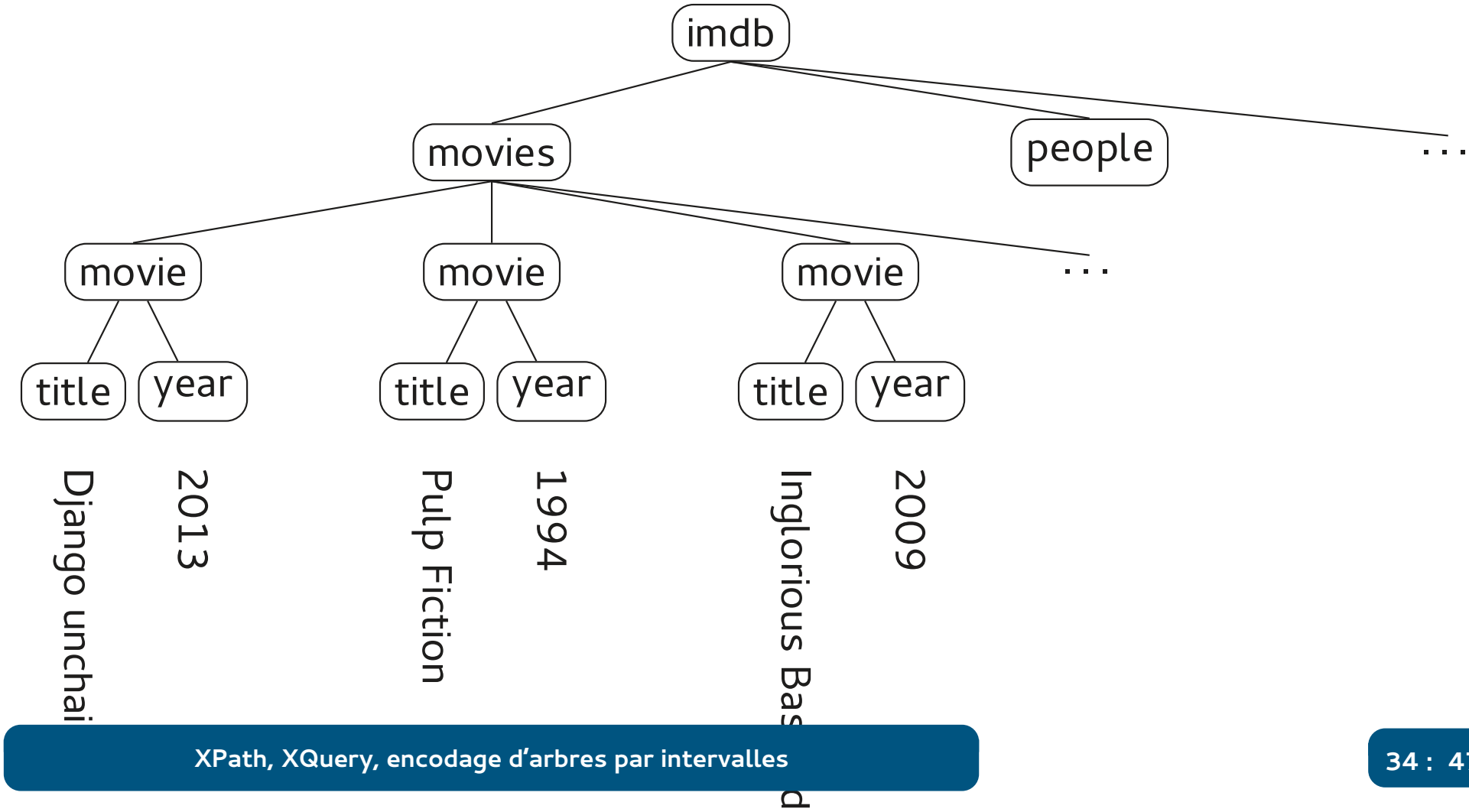
- Syntaxiquement, les motifs sont des types avec des variables de capture
- On utilise des automates avec marquage
- Théorème (typage exact du filtrage)
  Pour tout motif p, on peut calculer l'ensemble $\lbag p \rbag$ des valeurs qui ne font pas échouer le filtrage. De plus, $\lbag p \rbag$ est un type.
- On peut déterminer de manière exacte l'exhaustivité ou la redondance lors du filtrage :
  - $t \not\leq \lbag p_1 \rbag \vee \ldots \vee \lbag p_n \rbag$: filtrage redondant
  - $\lbag p_1 \rbag \vee \ldots \vee \lbag p_n \rbag \not\leq t$: filtrage non exhaustif (et production d'un contre-exemple)

4. XPath, XQuery, encodage d'arbres par intervalles

<title>XPath</title>

Langage de selection de nœeuds dans un arbre (W3C).
Exemple de requête :

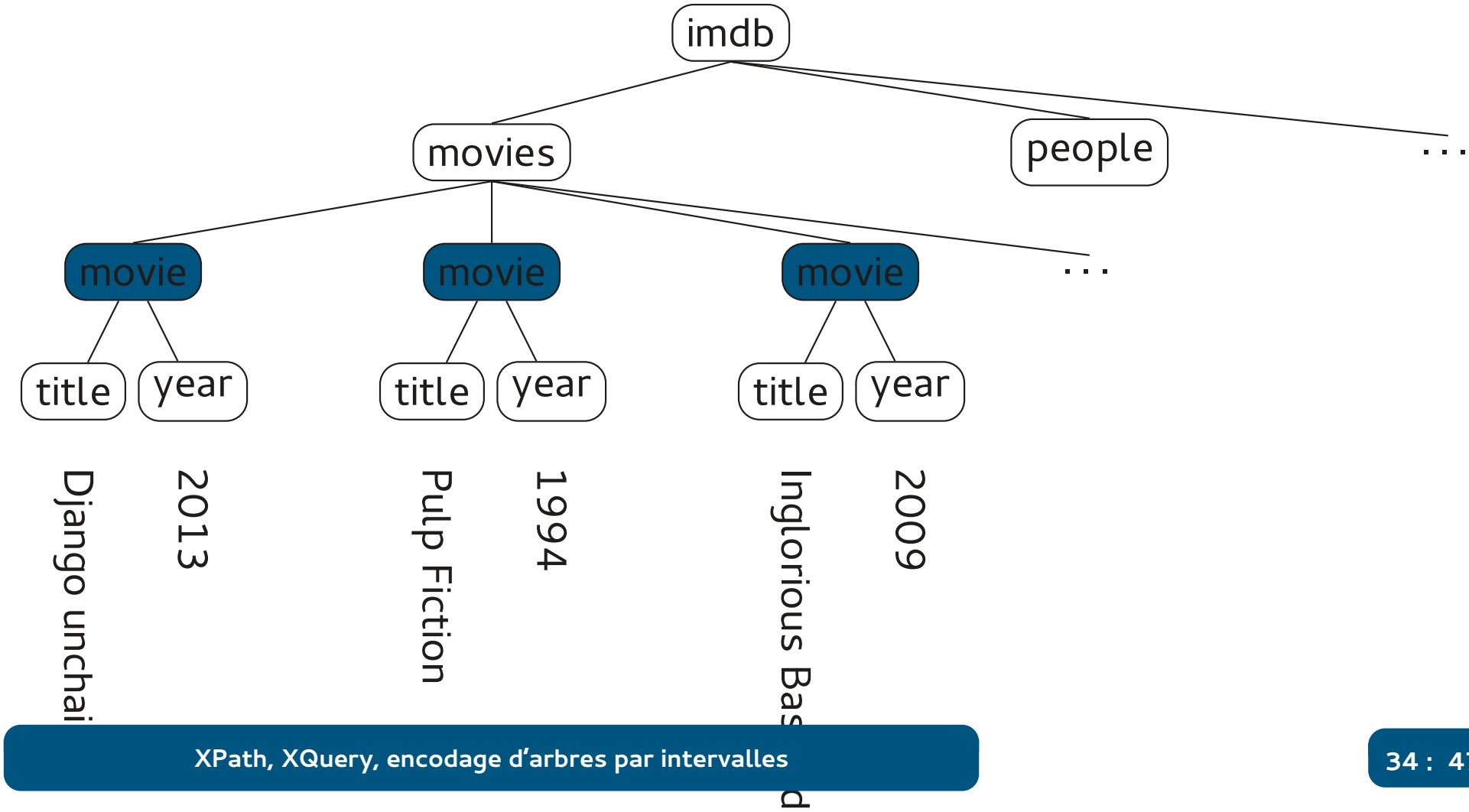
```
/descendant::movie[ child::year >= 2000 ]/child::title
/descendant::*[ self::year >= 2000 ]/parent::*/child::title
```



<title>XPath</title>

Langage de selection de nœeuds dans un arbre (W3C).
Exemple de requête :

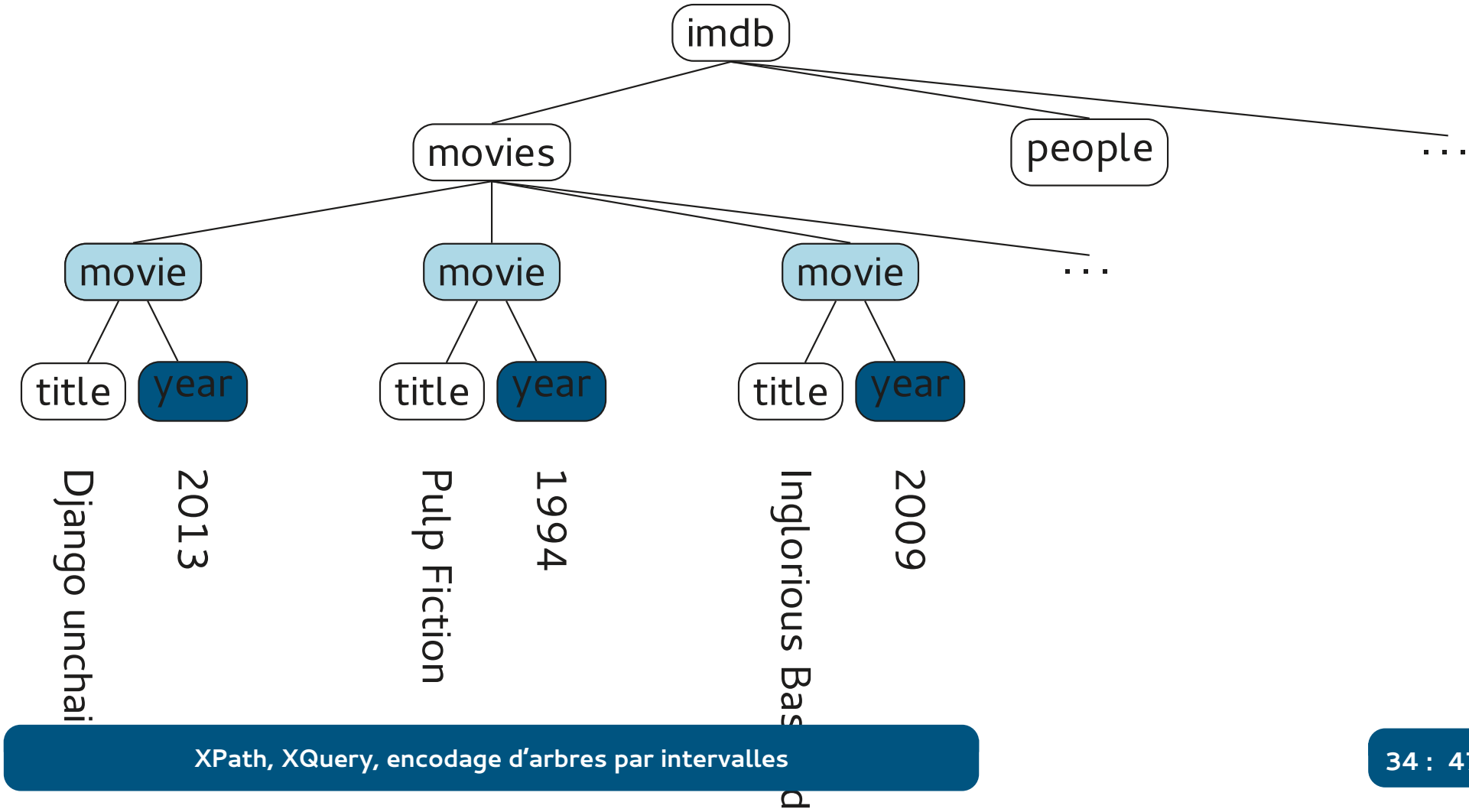
```
/descendant::movie[ child::year >= 2000 ]/child::title
/descendant::*[ self::year >= 2000 ]/parent::*/child::title
```



<title>XPath</title>

Langage de selection de nœeuds dans un arbre (W3C).
Exemple de requête :

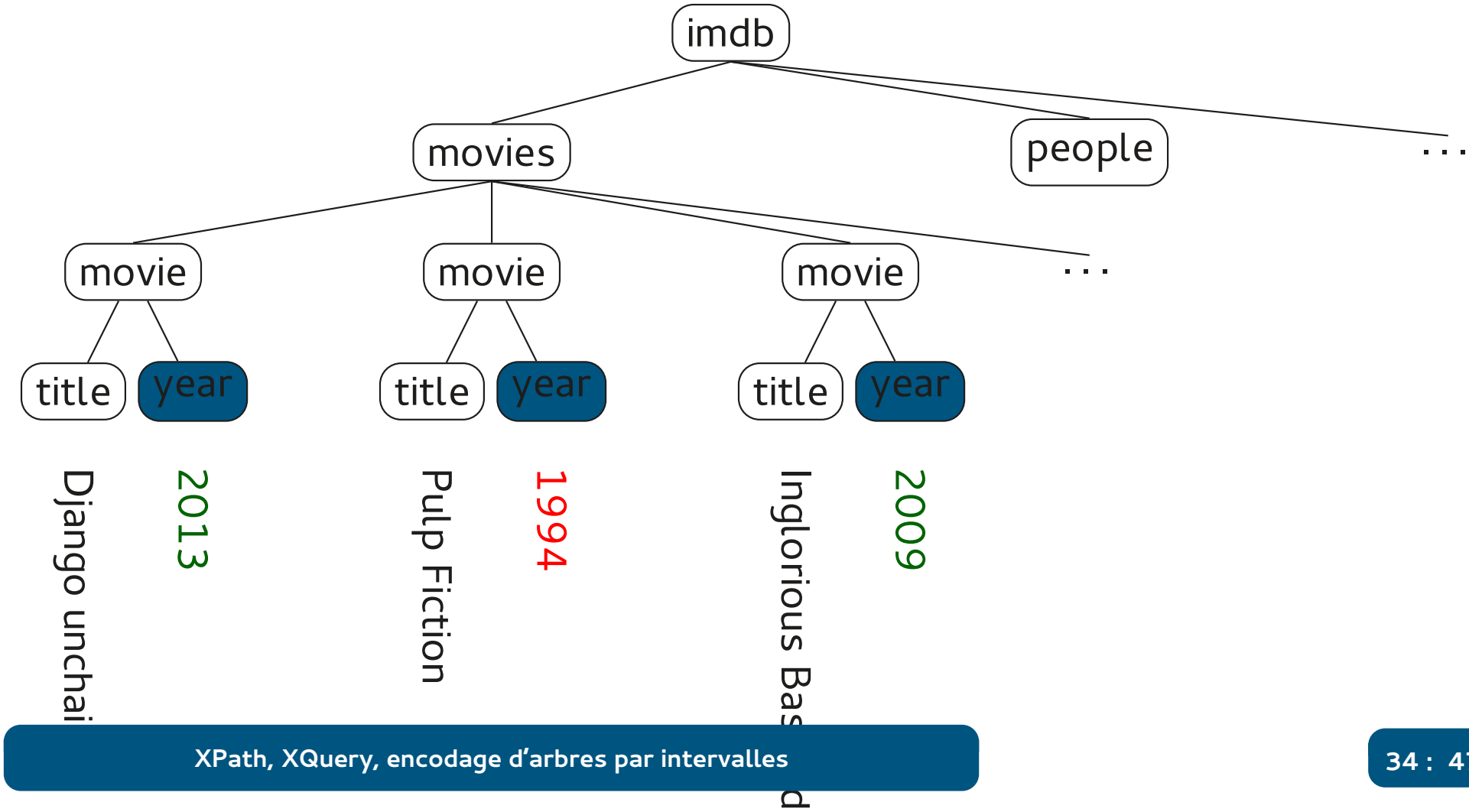
```
/descendant::movie[ child::year >= 2000 ]/child::title
/descendant::*[ self::year >= 2000 ]/parent::*/child::title
```



<title>XPath</title>

Langage de selection de nœeuds dans un arbre (W3C).
Exemple de requête :

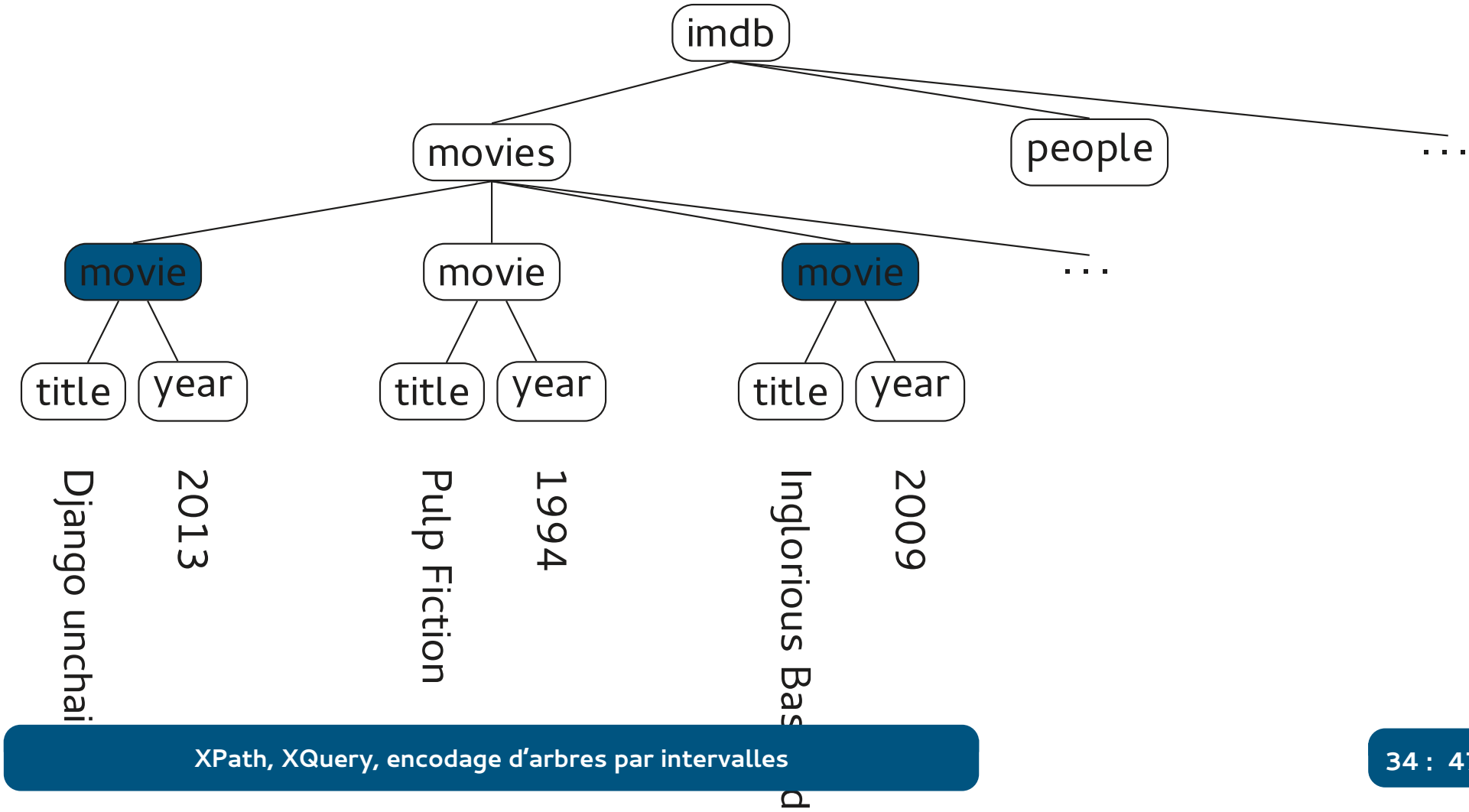
```
/descendant::movie[ child::year >= 2000 ]/child::title
/descendant::*[ self::year >= 2000 ]/parent::*/child::title
```



<title>XPath</title>

Langage de selection de nœeuds dans un arbre (W3C).
Exemple de requête :

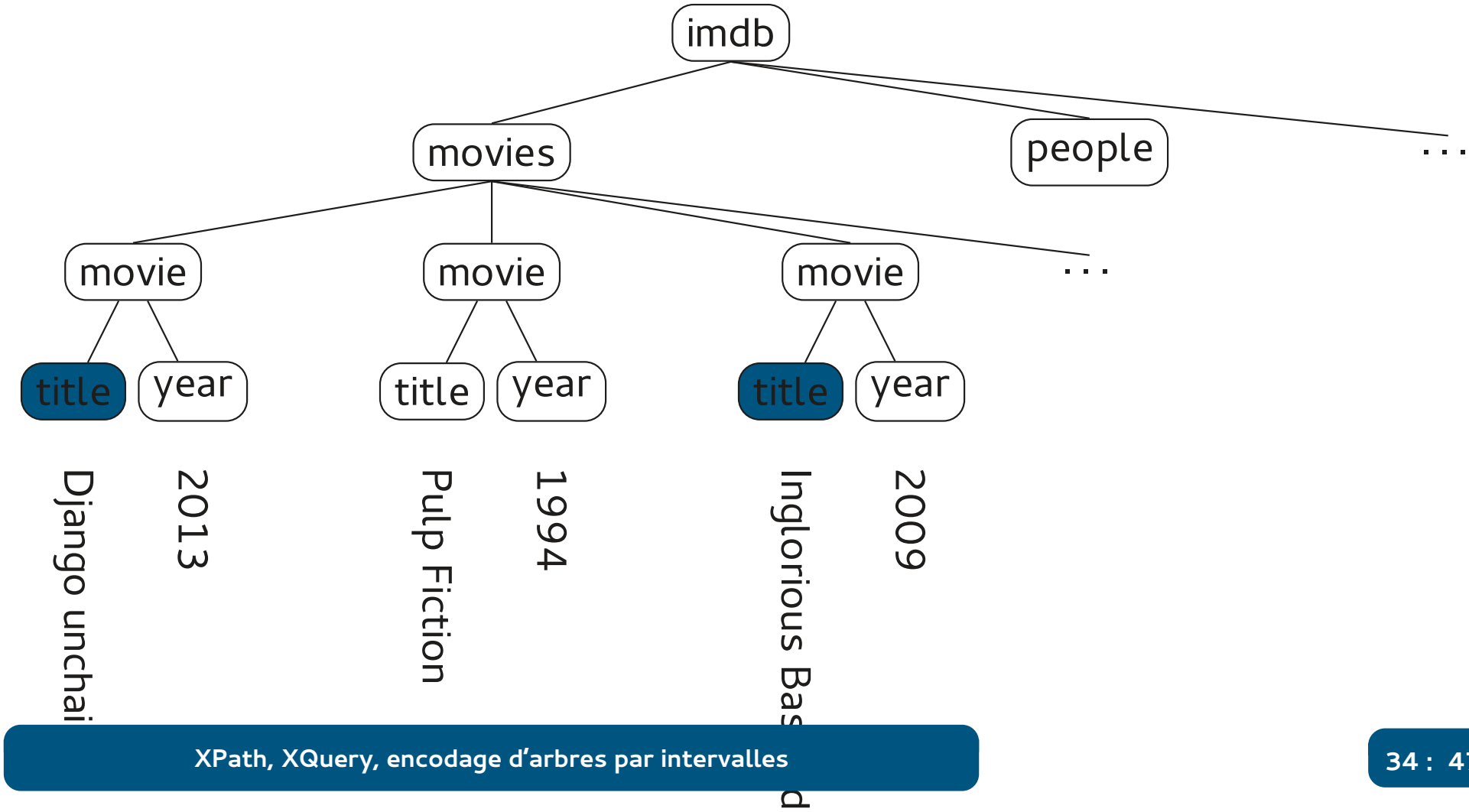
```
/descendant::movie[ child::year >= 2000 ]/child::title
/descendant::*[ self::year >= 2000 ]/parent::*/child::title
```



<title>XPath</title>

Langage de selection de nœeuds dans un arbre (W3C).
Exemple de requête :

```
/descendant::movie[ child::year >= 2000 ]/child::title
/descendant::*[ self::year >= 2000 ]/parent::*/child::title
```



<title>XQuery</title>

Langage de requête et transformation (W3C).
Exemple de programme :

```
let $doc := document("imdb.xml")
return <films>{
 for $i in $doc/descendant::movie[child::year >= 2000]/child::title
 return <titre>{$i}</titre>
}</films>
```

Produit :

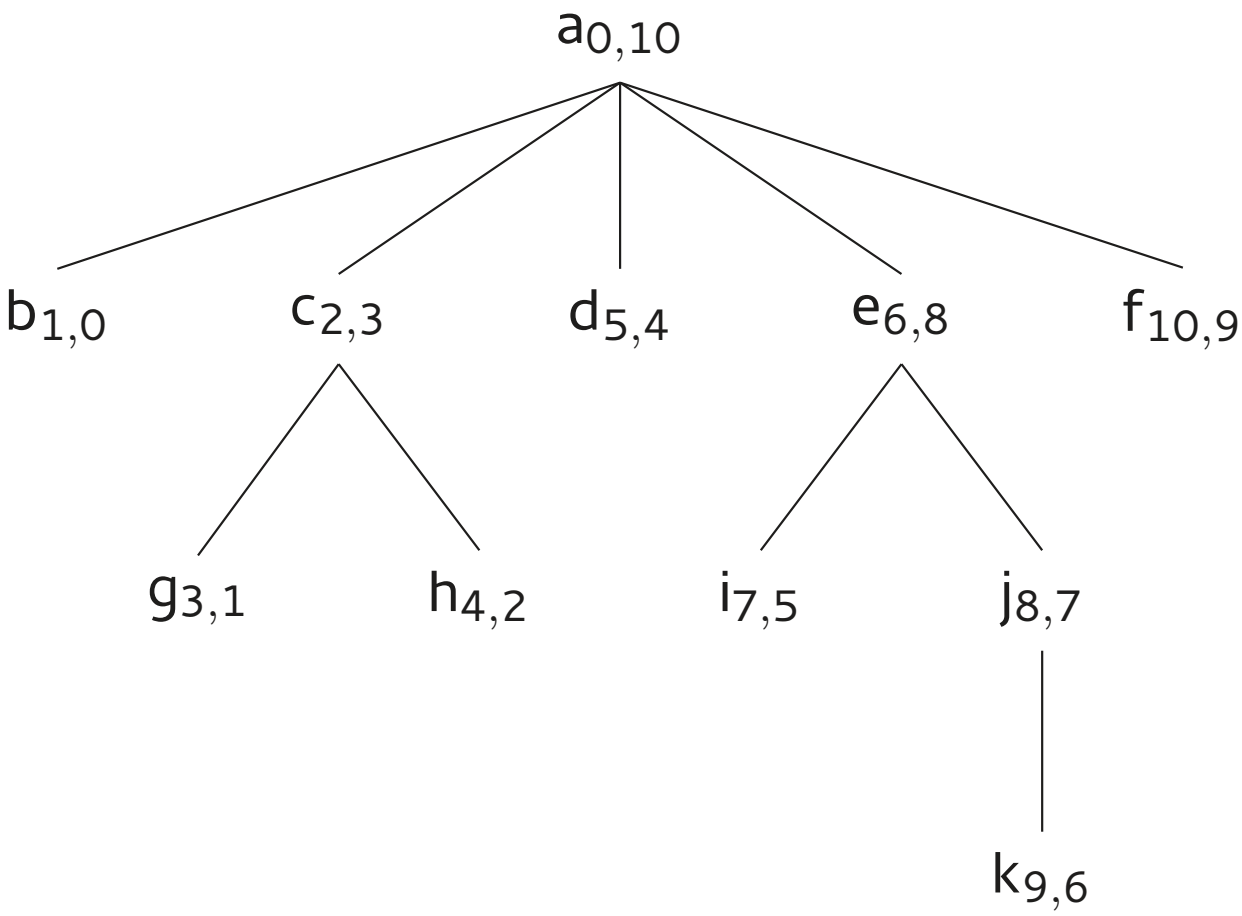
```
<films>
   <titre>Django Unchained</titre>
   <titre>Inglorious Basterds</titre>
</films>
```

<title>Problèmes avec XPath/XQuery</title>

- Le W3C dit quoi faire, pas comment
- Évaluation naïve inefficace
- Pas ou peu d'analyse statique/typage

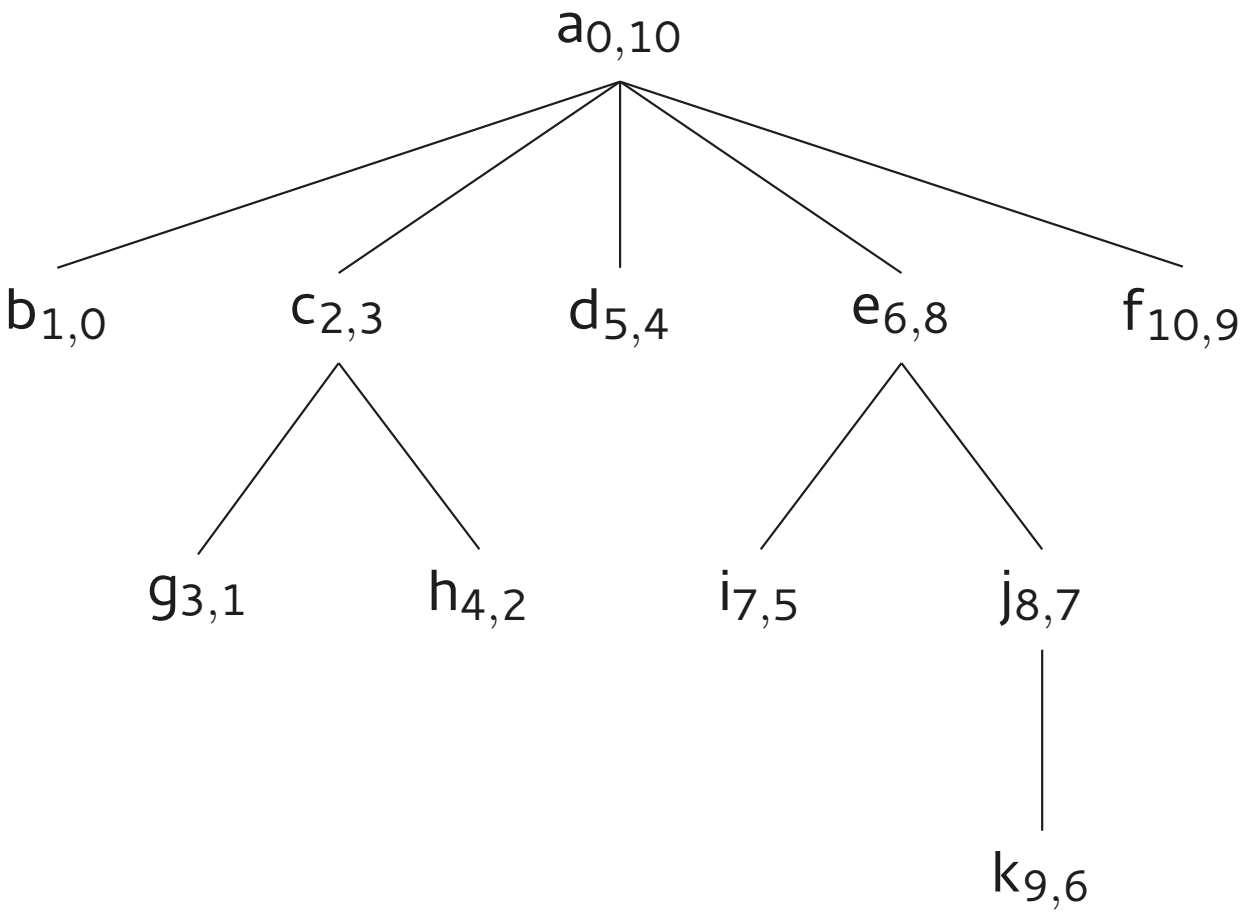
<title>Encodage d'un arbre en intervalles</title>

1. Permet de calculer efficacement des chemins XPath
2. Encodage compatible avec les BD relationnelles



<title>Encodage d'un arbre en intervalles</title>

1. Permet de calculer efficacement des chemins XPath
2. Encodage compatible avec les BD relationnelles



$descendant(x,y) =$
$x.pre < y.pre \wedge$
$x.post > y.post$

$ancestor(x,y) =$
$descendant(y,x)$

$child(x,y) =$
$descendant(x,y) \wedge$
$level(x) = level(y)-1$

<title>Encodage d'un arbre en intervalles</title>

1. Évaluation d'une requête XPath en $O(|Q| \times |T|)$
2. Traduction immédiate d'XPath en SQL :

```
SELECT DISTINCT z FROM root doc, x doc, y doc, z doc WHERE
  root.pre = 0 AND
  x.pre > root.pre AND x.post < root.post
  AND x.label = "movie" AND
  y.pre > x.pre AND y.post < x.post AND y.level - 1 = x.level
  AND y.label = "year" AND y.value >= 2000
  z.pre > x.pre AND z.post < x.post AND z.level - 1 = x.level
  AND z.label = "title"
ORDER BY z.pre
```

3. Base des évaluateurs XPath/XQuery modernes
4. Variantes avec des indices non contigus pour permettre l'insertion/suppression de nœuds

<title>Inclusion de requêtes</title>

Étant données deux requêtes XPath R_1 et R_2 on veut savoir si :

$$R_1 \subseteq R_2$$

pour tout document d'entrée

- détecter du code mort (prendre une R_2 vide)
- maintenance de vue/cache
- ...

On peut utiliser l'encodage pre/post pour l'analyse statique :

- Transforme la requête XPath en formule logique du 1er ordre + arithmétique entière
- On donne le tout à un solveur SMT *[Benedikt, Cheney, 2010]*
- On attend (et on espère)

5. Logiques pour les arbres

<title>Logiques d'arbres</title>

Sur quel langage de requêtes sur les arbres peut-on raisonner :

XPath trop verbeux, pas de bonnes propriétés de clotures

Automates d'arbres trop « bas-niveau » (récursion explicite)

Formules logiques oui mais lesquelles ?

Remarque : on veut conserver le pouvoir expressif des langages réguliers (clotures par opérations booléenes, implantations efficaces, langage utilisé pour les types, . . .)

<title>Logique du 1er ordre sur les arbres</title>

$$
\begin{array}{rcll}
\phi & ::= & U(x) & \text{(relation unaire)} \\
 & | & B(x,y) & \text{(relation binaire)} \\
 & | & \phi \vee \phi & \text{(disjonction)} \\
 & | & \exists x.\phi & \text{(quant. existentielle)} \\
 & | & \top & \text{(vrai)}
\end{array}
$$

avec le sucre syntaxique usuel : $\phi \wedge \phi$, $\forall x.\phi$, $\phi \Rightarrow \phi$, ...

Exemple : « tous les nœuds a ont un fils b »

$$\forall x.(\mathtt{label_a}(x) \Rightarrow \exists y.\mathtt{child}(x,y) \wedge \mathtt{label_b}(y))$$

<title>Logique du 1er ordre sur les arbres</title>

$$
\begin{array}{rcll}
\phi & ::= & U(x) & \text{(relation unaire)} \\
 & | & B(x,y) & \text{(relation binaire)} \\
 & | & \phi \vee \phi & \text{(disjonction)} \\
 & | & \exists x.\phi & \text{(quant. existentielle)} \\
 & | & \top & \text{(vrai)}
\end{array}
$$

avec le sucre syntaxique usuel : $\phi \wedge \phi$, $\forall x.\phi$, $\phi \Rightarrow \phi$, ...
Exemple : « tous les nœuds a ont un fils b »

$$\forall x.(\mathtt{label_a}(x) \Rightarrow \exists y.\mathtt{child}(x,y) \wedge \mathtt{label_b}(y))$$

Limites du 1er ordre ?
« L'ensemble des arbres qui ont un nombre pair de a » n'est pas FO.

<title>Logique du 1er ordre sur les arbres</title>

$$
\begin{array}{rcll}
\phi & ::= & U(x) & \text{(relation unaire)} \\
 & | & B(x,y) & \text{(relation binaire)} \\
 & | & \phi \vee \phi & \text{(disjonction)} \\
 & | & \exists x.\phi & \text{(quant. existentielle)} \\
 & | & \top & \text{(vrai)}
\end{array}
$$

avec le sucre syntaxique usuel : $\phi \wedge \phi$, $\forall x.\phi$, $\phi \Rightarrow \phi$, ...
Exemple : « tous les nœuds a ont un fils b »

$$\forall x.(\mathtt{label}_{\mathtt{a}}(x) \Rightarrow \exists y.\mathtt{child}(x,y) \wedge \mathtt{label}_{\mathtt{b}}(y))$$

Limites du 1er ordre ?
« L'ensemble des arbres qui ont un nombre pair de a » n'est pas FO.
(Exercice : écrire l'automate d'arbre correspondant)

<title>Logique Monadique du Second ordre</title>

$$
\begin{array}{rcll}
\phi & ::= & \ldots & (1^{\text{er}} \text{ ordre}) \\
 & | & x \in X & (\text{relation unaire variable}) \\
 & | & \forall X.\phi & (\text{quant. univ. second-ordre}) \\
 & | & \exists X.\phi & (\text{quant. exist. second-ordre})
\end{array}
$$

Théorème :

$$\text{MSO-définissable} \equiv \text{langage régulier d'arbre}$$

<title>Logique Monadique du Second ordre</title>

$$
\begin{array}{rcll}
\phi & ::= & \ldots & (1^{\text{er}} \text{ ordre}) \\
 & | & x \in X & (\text{relation unaire variable}) \\
 & | & \forall X.\phi & (\text{quant. univ. second-ordre}) \\
 & | & \exists X.\phi & (\text{quant. exist. second-ordre})
\end{array}
$$

Théorème :

MSO-définissable $\equiv$ langage régulier d'arbre

Problème :

Pour tout entier n, il existe ϕ MSO-définissable telle que

$$
|A_\phi| = \left. 2^{2^{\cdots^{|\phi|}}} \right\} n \text{ fois}
$$

<title>μ-calcul sur les arbres</title>

$$
\begin{array}{rcll}
\phi & ::= & \top & \text{(vrai)} \\
 & | & l \mid \neg l & \text{(nom de nœud)} \\
 & | & \phi \wedge \phi & \text{(conjonction)} \\
 & | & \phi \vee \phi & \text{(disjonction)} \\
 & | & \downarrow_i \phi \mid \uparrow_i \phi \mid \neg \downarrow_i \top \mid \neg \uparrow_i \top & i \in \{1,2\} \text{ (modalité)} \\
 & | & \mu X.\phi & \text{(plus petit point fixe)}
\end{array}
$$

Exemple :

$$\uparrow_1 \mathtt{a} \wedge \mathtt{b} \wedge \mu X.(\mathtt{c} \vee \downarrow_2 X)$$

<title>μ-calcul sur les arbres</title>

$$
\begin{array}{rcll}
\phi & ::= & \top & \text{(vrai)} \\
 & | & l \mid \neg l & \text{(nom de nœud)} \\
 & | & \phi \wedge \phi & \text{(conjonction)} \\
 & | & \phi \vee \phi & \text{(disjonction)} \\
 & | & \downarrow_i \phi \mid \uparrow_i \phi \mid \neg \downarrow_i \top \mid \neg \uparrow_i \top & i \in \{1,2\} \text{ (modalité)} \\
 & | & \mu X.\phi & \text{(plus petit point fixe)}
\end{array}
$$

Exemple :

$$\uparrow_1 \mathrm{a} \wedge \mathrm{b} \wedge \mu X.(\mathrm{c} \vee \downarrow_2 X)$$

- Équivalent aux langages réguliers
- SAT(ϕ) décidable en $O(2^{|\phi|})$ (seulement !)
- Implantation efficace :
  `http://wam.inrialpes.fr/web-solver/webinterface.html`

[Genevès et. al. 2005,2011]

Conclusion

<title>XML ?</title>

Au final, un bon pretexte pour s'intéresser à plein de choses :

- Langages réguliers, automates
- Logiques (de toutes sortes), expressivité, complexité
- Représentations efficaces (encodage d'arbres, BDD, . . .)
- Langages de programmation (compilation efficace de requête XML)
- . . .

Ça permet aussi de voir que toutes ces choses sont reliées

<title>1999, l'année d'XML* ?</title>

$\star$: source, PC Impact, décembre 1998.
XML n'est plus vraiment une techno. à la mode (sauf si on en fait dans le *cloud*). Il reste quand même plein de choses à faire :

- Structure de données efficaces avec mise à jour pour les arbres
- Évaluateurs efficaces basés sur des automates
- Populariser l'analyse statique dans la communauté BD
- Sous-typage sémantique + polymorphisme à la ML
- μ-calcul ou MSO + procédures de décisions
- ...