Compilation: Frontend

8 avril 2020

1 Préliminaires

1.1 Récursion/induction et point fixe

Dans ce cours, on verra plusieurs algorithmes écrits en pseudo-code. Pour plus de clarté et d'uniformité, on utilisera deux structures particulières : l'induction et le point fixe.

Induction. L'induction est le pendant algorithmique de la récursion en mathématique où l'on "récurse" pas seulement sur des entiers, mais sur une structure de donnée finie (entier, mot, arbre). Pour ceux qui ont fait du Haskell ou du OCaml, il s'agit d'une fonction récursive commençant par un "pattern-matching" sur un argument. Une induction est donc une fonction (souvent auxiliaire) avec trois points :

- l'argument inductif : il s'agit d'un argument ou sous-argument (si l'argument est un tuplet) qui va décroître à chaque appel,
- le ou les cas de base (0, ϵ, feuille...),
- le ou les cas inductifs ($(n+1, t.\omega$, noeud...) qui ont le droit de rappeler la fonction avec un argument inductif plus petit.

Le déroulement de l'algorithme est alors le suivant : on regarde l'argument inductif, si c'est l'un des cas de base, on exécute le code associé, et si c'est un cas inductif, on exécute le cas associé sauf que à chaque appel récursif on met en pause, on exécute l'algorithme sur ce cas plus petit, puis on revient a notre exécution et on utilise le résultat.

Exemple 1. *La taille d'un mot peut être calculée inductivement.*
Fonction $|\omega|$ calculée par induction :

- *argument inductif : mot ω,*
- $|\epsilon| = 0$
- $|t\omega| = 1 + |\omega|$

Remarque 1. *En pseudo-code, on se passera de discussion d'optimisations telles que la récursion terminale.*

Point fixe. Le principe d'un algorithme par point fixe est le suivant : on se donne un ensemble fini E et on cherche le plus petit sous-ensemble $R \subseteq E$ qui contienne un certain $I \subseteq E$ et qui vérifie une condition croissante $R = R \cup f(R)$. C'est pourquoi il faut donner :

- un ensemble fini E,
- une initialisation $I \subseteq E$, que l'on écrira $R := I$ avec dans l'idée que R est un mutable qui va être augmenté à chaque pas,
- un "pas" f, que l'on écrira $R := R \cup f(R)$ ou même $R := f(R)$ si f est croissant.

L'exécution associée est alors la suivante :

```
Set R := I;
Do {
  R' := union(R, f(R));
  fixe := isEmpty(R' - R);
  R := R'
} While (not fixe)
```

Exemple 2. *Soit E un ensemble fini, $A \subseteq E$ et f une fonction de E dans les parties de E. La clôture de A par f (qu'on note $f^*(A)$) est l'ensemble des éléments de E accessibles depuis A par des applications successives de f. On la génère par point fixe :*

- *Ensemble : E*
- *Initialisation : $f^*(A) := A$,*
- *Pas : $f^*(A) := f^*(A) \cup \{x \in E \mid \exists a \in f^*(a), x \in f(a)\}$.*

Remarque 2. *La notion peut être généralisée à d'autres structures finies que les sous-ensembles d'un ensemble fini, cela marche pour tout ensemble fini ordonné. On peut même se passer de la condition "fini" en mettant des conditions plus faibles (dcpo, coinductifs), mais on doit alors procéder avec une exécution paresseuse et on n'a pas accès à certaines informations.*

1.2 Tokens

On a vu, dans l'introduction et dans la description globale du compilateur, que les lettres de nos mots n'étaient pas des lettres, mais des "tokens", c'est à dire des lettres avec un contenu :

Définition 1.
Un alphabet de tokens $\boldsymbol{\Sigma}$ est la donnée d'un alphabet Σ et, pour chaque lettre $t \in \Sigma$, d'un ensemble $\texttt{tokens}(t)$.
Un token $\boldsymbol{t} \in \boldsymbol{\Sigma}$ est un couple (t, T) pour $t \in \Sigma$ et $T \in \texttt{tokens}(t)$.

Notations 1.
On utilisera toujours version bleu et grasse $\boldsymbol{t}$ pour désigner un token et la version noir t pour la lettre sous-jacente. De même, Σ est l'alphabet sous-jacent à l'alphabet de token $\boldsymbol{\Sigma}$, et ω est le mot de lettres sous-jacent au mot de tokens $\boldsymbol{\omega}$.

Remarque 3. *La plupart du temps, on n'a pas besoin de distinguer le token de la lettre sous-jacente. En fait, le contenu du token ne sera utilisé que pour les actions des automates "shift-action".*

Toutefois, il est impératif de ne jamais parler de token dans les règles de vos (variantes d') automates, car un alphabet de token peut contenir une infinité de token !

Dans ce cadre, on n'explicitera pas les token, même pour les études de complexité (ce devrait être nécessaire pour la complexité en espace, mais on ne pourrait pas dire grand chose sans spécifier le contenu des tokens).

Définition 2.
Un langage sur un alphabet Σ est simplement un ensemble de mots $L \subseteq \Sigma^*$.

Par contre on ne parlera jamais de langage de token : dans le frontend, on ne différencie jamais deux tokens par autre chose que le symbole associé. Par abus de langage, on utilisera des fois la notation $\boldsymbol{L} \subseteq \boldsymbol{\Sigma}^*$, pour parler d'un langage $L \subseteq \Sigma^*$ et de l'ensemble $\boldsymbol{L} := \{\boldsymbol{\omega} \mid \omega \in L\}$.

2 Grammaires

2.1 Définitions

Définition 3.
Une "grammaire hors contexte" est donnée par
- *Un ensemble $\mathcal{N}$ de non-terminaux,*
- *Un ensemble (disjoint) $\boldsymbol{\mathcal{T}}$ de terminaux,*
- *Un non-terminal principal $S \in \mathcal{N}$,*
- *Pour chaque non-terminal $N \in \mathcal{N}$, un ensemble $\tau(N) \subseteq (\mathcal{T} \uplus \mathcal{N})^*$ de mots composés de terminaux et non-terminaux, appelés règles de N.*

Dans la suite, on appellera "grammaire" une grammaire hors contexte.

Notations 2 (Mathématique)**.**
Pour poser la théorie et faire les preuves, on utilisera les notations suivantes :
- *les non-terminaux sont dénotés par les métavariables majuscules $S, N, ...$,*
- *les terminaux sont dénotés par les métavariables minuscules $\boldsymbol{s}, \boldsymbol{t}...$ qui peuvent correspondre à des tokens, dans les exemples et exercices, on utilisera aussi de symboles $(+, *, \lambda..)$ pour plus de lisibilité (attention le epsilon ϵ ne sont pas utilisés),*
- *les symboles qui peuvent être terminaux ou non-terminaux[1] sont dénotés par les métavariables minuscules soulignées $\underline{a}, \underline{b}, \underline{c}...$,*
- *le non-terminal principal est toujours S,*
- *les règles sont notées $N \mapsto \omega$ pour $\omega \in \tau(N)$,*
- *le symbole ϵ est spécial : il désigne le mot vide comme résultat de la règle.*

1. Lors de la description des algorithmes on peut décrire des étapes qui se passent pareil sur des terminaux et non terminaux

Notations 3 (Informatique)**.**
Pour décrire une grammaire formelle (réelle), on utilisera les notations suivantes :

- *les non-terminaux sont dénotés par des noms entre crochets* `<expression>`*,* `<toto>`*... commençant par des minuscules,*
- *les terminaux sont dénotés par des mots-clefs verbatim majuscule* `<VAR>`*,* `<TOTO>`*..., des caractères spéciaux verbatim* `£,(,),+`*...,*
- *lorsque* $\tau(N) = \{\omega_1, ..., \omega_n\}$*, les règles sont notées* `<non-term>` `:=` ω_1 `|` $\cdots$ `|` ω_n
- *le non-terminal principal est celui dont les règles sont écrites en premières*
- *attention : le mot vide est ici représenté par une absence de mot et non pas par* ϵ*.*

Notations 4 (Exercices)**.** *Pour faire des exercices, on se permettra de mélanger les deux types de notations tant qu'il n'y a pas d'ambiguïté.*

Exemple 3. *La grammaire du lambda calcul polymorphe (système F) est la suivante :*

$$\begin{array}{ll} S \mapsto \boldsymbol{x} & T \mapsto \boldsymbol{t} \\ S \mapsto \lambda \boldsymbol{x} : T.S & T \mapsto T \to T \\ S \mapsto \Lambda \boldsymbol{t}.S & T \mapsto T \times T \\ S \mapsto S\ S & T \mapsto (T) \\ S \mapsto S\ T & \\ S \mapsto (S) & \end{array}$$

où $\boldsymbol{x}$ *et* $\boldsymbol{t}$ *représentent des terminaux encapsulant un token (les variables de termes pour le premier, et de type pour le second). Remarquez que seuls* S *et* T *sont des non terminaux, tous les symboles* (λ, ., :, $\to$, *times et même les parenthèses) sont des terminaux.*

De façon plus pratique, on peut utiliser une formulation informatique :

```
<terme> := <VAR>
         | \<VAR>:<type>.<terme>
         | <LAMBDA><VARTYPE>.<terme>
         | <terme><terme>
         | <terme><type>
         | (<terme>)
```

```
<type> := <VARTYPE>
        | <type> <FLECHE> <type>
        | <type> * <type>
        | (<type>)
```

Pour un exercice on se permetra de mélangerles notation pour aller au plus

lisible, par exemple :

$$
\begin{aligned}
\texttt{<terme>} := \ & \boldsymbol{x} \\
& \mid \lambda \boldsymbol{x} : \texttt{<type>}.\texttt{<terme>} \\
& \mid \Lambda \boldsymbol{\alpha}.\texttt{<terme>} \\
& \mid \texttt{<terme><terme>} \\
& \mid \texttt{<terme><type>} \\
& \mid \texttt{(<terme>)}
\end{aligned}
\qquad
\begin{aligned}
\texttt{<type>} := \ & \boldsymbol{\alpha} \\
& \mid \texttt{<type>} \to \texttt{<type>} \\
& \mid \texttt{<type>} \times \texttt{<type>} \\
& \mid \texttt{(<type>)}
\end{aligned}
$$

Dans tous les cas, la résolution est la même à renomage près, et vous pouvez utiliser une autre représentation sur votre copier si vous le souhaitez ***à condition d'indiquer votre renomage en début d'exercice.***

Remarquez que pour chacun des cas, il faut préciser ce que sont les tokens $\boldsymbol{x}$ *(ou* `<VAR>`*) et* $\boldsymbol{t}$ *(ou* `<VARTYPE>` *ou* $\boldsymbol{\alpha}$*). Ici il s'agit de tokens contenant des chaines de caractères.*

Définition 4 (Règles partielles)**.**
Soit $\mathcal{G} = (\mathcal{T}, \mathcal{N}, S, \tau)$ *une grammaire. On appelle "règles partielles de* $\mathcal{G}$ *depuis un non-terminal* $N \in \mathcal{N}$*", l'ensemble*

$$\tau_\bullet(N) = \{(\omega_1, \omega_2) \mid \omega_1\omega_2 \in \tau(N)\} \subseteq (\mathcal{T} \uplus \mathcal{N})^* \times (\mathcal{T} \uplus \mathcal{N})^*.$$

On dénote $N \mapsto \omega_1 \bullet \omega_2$ *ces règles partielles* $(\omega_1, \omega_2) \in \tau_\bullet(N)$*.*
On qualifie d' "initiale" une règle partielle de la forme $N \mapsto \bullet\omega$ *et de "finale" une règle partielle de la forme* $N \mapsto \omega\bullet$*.*

Définition 5 (Multistep)**.**
Soit $\mathcal{G} = (\mathcal{T}, \mathcal{N}, S, \tau)$ *une grammaire.*
On appelle "réduction multistep" les couples $\omega \mapsto^* \omega'$ *pour* $\omega, \omega' \in (\mathcal{N} \uplus \mathcal{T})^*$ *définis comme la clôture transitive réflexive de* $\mapsto$ *:*

- *Ensemble :* $(\mapsto^*) \subseteq ((\mathcal{N} \uplus \mathcal{T})^*)^2$*,*
- *Initialisation :* $\omega \mapsto^* \omega$ *pour tout* $\omega \in \mathcal{N} \uplus \mathcal{T}$*, et* $N \mapsto^* \omega$ *lorsque* $N \mapsto \omega$*,*
- *Pas :* $\omega_1\omega_2 \mapsto^* \omega_1'\omega_2'$ *lorsque* $\omega_1 \mapsto \omega_1'$ *et* $\omega_2 \mapsto \omega_2'$*, de plus* $\omega \mapsto^* \omega''$ *lorsqu'il existe* $\omega \mapsto^* \omega' \mapsto^* \omega''$*.*

Remarque 4. *Ceci n'est pas un algorithme car l'ensemble* $((\mathcal{N} \uplus \mathcal{T})^*)^2$ *est infini. Ce point-fixe montre par contre que la relation* $(\mapsto^*)$ *est récursivement énumérable, c'est à dire que si* $\omega \mapsto^* \omega'$*, on le trouvera en un temps fini.*

Définition 6 (Langage d'une grammaire)**.**
Le langage $L_\mathcal{G}$ *d'une grammaire* $\mathcal{G} = (\mathcal{T}, \mathcal{N}, S, \tau)$ *est l'ensemble des mots de terminaux accessibles depuis le non-terminal initial :*

$$L_\mathcal{G} := \{\omega \in \mathcal{T}^* \mid S \mapsto^* \omega\} \ .$$

On dit qu'un mot $\boldsymbol{\omega} \in \mathcal{T}^*$ *est reconnu par* $\mathcal{G}$ *si* $\omega \in L_\mathcal{G}$*.*

2.2 Arbres syntactiques

Définition 7 (Arbre syntactique version mathématique)**.**
Soit $\mathcal{G} = (\mathcal{T}, \mathcal{N}, S, \tau)$ une grammaire.
On définit les arbres syntactiques des non-terminaux de $\mathcal{G}$ comme un ensemble $\boldsymbol{\mathcal{N}}$ de tokens sur $\mathcal{N}$ définit par point fixe :

- *Ensemble : $\boldsymbol{\mathcal{N}} \subseteq \mathcal{N} \times (\mathcal{T} \uplus \boldsymbol{\mathcal{N}})^*$,*
- *Initialisation : $\boldsymbol{\mathcal{N}} = \emptyset$,*
- *Pas :*

$$\boldsymbol{\mathcal{N}} := \bigcup_{N \mapsto \underline{c_1} \ldots \underline{c_n}} \{(N, \underline{\boldsymbol{c_1}} \ldots \underline{\boldsymbol{c_n}}) \mid \forall i,\ \underline{c_i} \in \mathcal{T} \Rightarrow \underline{\boldsymbol{c_i}} \in \boldsymbol{\mathcal{T}} \quad et \quad \underline{c_i} \in \mathcal{N} \Rightarrow \underline{\boldsymbol{c_i}} \in \boldsymbol{\mathcal{N}}\}$$

On appelle arbre syntactique de $\mathcal{G}$ un arbre syntactique $\boldsymbol{\mathcal{S}}$ de $\mathcal{S}$.

Notations 5. *Un arbre syntactique $\boldsymbol{N} = (N, \underline{\boldsymbol{c_1}} \ldots \underline{\boldsymbol{c_n}})$ peut se représenter sous la forme d'un arbre de la racine labellé par N et avec pour fils les arbres syntaxiques $\boldsymbol{c_1} \ldots \boldsymbol{c_n}$.*

Définition 8 (Arbre syntactique version informatique)**.**
Soit $\mathcal{G} = (\mathcal{T}, \mathcal{N}, S, \tau)$ une grammaire.
On définit les arbres syntactiques `NonTerm` *partant de* `<non-term>` $\in \mathcal{N}$
`<non-term> := <a_1_1>...<a_1_k1> | ⋯ | <a_n_1>...<a_n_kn>`
différemment selon le langage :

- *comme un type-union (ex : en C) qui est une* struct *obtenue comme l'union des* structs `NonTerm1...NonTermN` *suivantes :*
  ```
  NonTermI := {a_i_1 : AI1, ... , a_i_ki : AIKI }
  ```
- *comme le type algébrique (ex : OCaml, Haskell) :*
  ```
  NonTerm :=  Regle1 of (A11,...,A1K1)
            | ...
            | RegleN of (AN1,...,ANKN)
  ```
- *comme une classe abstraite (ex : Java) hérité par n versions concrétisés :*
  ```
  public class NonTermI extends NonTerm {
    AI1 a_i_1;
    ...
    AIKI a_i_ki;
  }
  ```

Remarque 5. *De part la multiplicité des implémentations, on utilisera toujours la version mathématique. De plus, la version mathématique est, par définition, restreinte aux arbres finis, ce qui nous est nécessaire pour le théorème suivant :*

Proposition 1 (Induction sur les arbres syntactique)**.**
Soit $\mathcal{G} = (\mathcal{T}, \mathcal{N}, S, \tau)$ une grammaire.
Il y a un ordre bien fondé sur les tokens de terminaux et les non-terminaux tel que pour tout $\boldsymbol{N} = (N, \boldsymbol{c_1} \ldots \boldsymbol{c_n}) \in \boldsymbol{\mathcal{N}}$ et tout $i \leq n$, $\boldsymbol{c_i} < \boldsymbol{N}$.
Autrement dit, on peut faire une induction sur les arbres syntactiques.

Définition 9 (Mot reconnu par un arbre syntactique)**.**
Soit $\mathcal{G} = (\boldsymbol{\mathcal{T}}, \mathcal{N}, S, \tau)$ une grammaire et $\boldsymbol{N} \in \boldsymbol{\mathcal{N}}$ un arbre syntactique.
Le mot $\mathtt{mot}(\boldsymbol{c}) \in \boldsymbol{\mathcal{T}}^*$ reconnu par $\underline{\boldsymbol{c}} \in \boldsymbol{\mathcal{N}} \uplus \boldsymbol{\mathcal{T}}$ est définit par induction :
- *argument inductif : $\underline{\boldsymbol{c}} \in \boldsymbol{\mathcal{N}} \uplus \boldsymbol{\mathcal{T}}$,*
- *cas $\boldsymbol{t} \in \boldsymbol{\mathcal{T}}$: alors $\mathtt{mot}(\boldsymbol{t}) = \boldsymbol{t} \in \boldsymbol{\mathcal{T}}^*$,*
- *cas $\boldsymbol{N} = (N, \boldsymbol{c_1 ... c_n}) \in \boldsymbol{\mathcal{N}}$: alors $\mathtt{mot}(\boldsymbol{N}) = \mathtt{mot}(\boldsymbol{c_1}) ... \mathtt{mot}(\boldsymbol{c_n})$.*

Proposition 2.
Soit $\mathcal{G} = (\boldsymbol{\mathcal{T}}, \mathcal{N}, S, \tau)$ une grammaire et $N \in \mathcal{N}$ un non-terminal.
L'ensemble des mots reconnus par des arbres syntaxiques issus de N est exactement l'ensemble des réduits terminaux de N :

$$N \mapsto^* \omega \quad \Leftrightarrow \quad \exists \boldsymbol{N},\ \boldsymbol{\omega} = \mathtt{mot}(\boldsymbol{N})$$

Définition 10 (Ambigüité)**.**
Une grammaire $\mathcal{G} = (\boldsymbol{\mathcal{T}}, \mathcal{N}, S, \tau)$ est ambigüe si un non terminal a plusieurs arbres syntactiques reconnaissant un même mot, c'est à dire si :

$$\exists N \in \mathcal{N},\ \exists (N, \boldsymbol{u}) \neq (N, \boldsymbol{v}) \in \boldsymbol{\mathcal{N}}, \quad \mathtt{mot}(N, \boldsymbol{u}) = \mathtt{mot}(N, \boldsymbol{v})\ .$$

On peut associer des règles d'associativité et de priorité sur une grammaire ambigüe pour la désambigüiser.

Définition 11 (Paire critique)**.**
Soit $\mathcal{G} = (\boldsymbol{\mathcal{T}}, \mathcal{N}, S, \tau)$.
Une paire critique est la donnée de deux règles $(N \mapsto u)$, $(N \mapsto v)$ sur un même non-terminal N, et de deux décompositions différentes $u = u_1 N u_2$ et $v = v_1 N v_2$ telles que $u_1 v_1 = v_1 u_1$ et $u_2 v_2 = v_2 u_2$.
On note $(N \mapsto \underline{N} u_2 \mid v_1 \underline{N} v_2)$ une telle paire critique.

Proposition 3. *Une grammaire avec une paire critique $(N \mapsto u_1 N u_2,\ N \mapsto v_1 N v_2)$ dont les règles sont productives est nécessairement ambigüe.*

Remarque 6. *Ce n'est pas une équivalence, càd. qu'il existe des grammaires ambigües sans paires critiques. Mais il s'agit de la majeur partie des cas.*

Définition 12 (Priorité)**.**
Soit $\mathcal{G} = (\boldsymbol{\mathcal{T}}, \mathcal{N}, S, \tau)$ une grammaire.
Une priorité est un ordre $(N \mapsto u) > (N \mapsto v)$ entre deux règles différentes sur un même non-terminal.
Une telle priorité va casser toutes les paires critiques $(N \mapsto u_1 \underline{N} u_2 \mid v_1 \underline{N} v_2)$ entre ces deux règles (càd. $u = u_1 N u_2$ et $v = v_1 N v_2$) en interdisant les arbres syntaxiques contenant un sous-arbre de la forme $(N, \boldsymbol{u_1}(N, \boldsymbol{v_1 N v_2})\boldsymbol{u_2})$ au profit de l'arbre équivalent de la forme $(N, \boldsymbol{v'_1}(N, \boldsymbol{u'_1 N u'_2})\boldsymbol{v'_2})$ où $\boldsymbol{v'_1 u'_1} := \boldsymbol{u_1 v_1}$ et $\boldsymbol{u'_2 v'_2} := \boldsymbol{v_2 u_2}$.

Remarque 7. *Étant donné deux règles $(N \mapsto u)$, $(N \mapsto v)$, il peut y avoir plusieurs décompositions $(u, v) = (u_1 N u_2, v_1 N v_2)$ avec $u_1 v_1 = v_1 u_1$ et $u_2 v_2 = v_2 u_2$. Dans ce cas, dire que la première règle est prioritaire sur la seconde impose la priorité pour toutes les décompositions.*

Définition 13 (Associativité)**.**
Soit $\mathcal{G} = (\boldsymbol{\mathcal{T}}, \mathcal{N}, S, \tau)$ une grammaire.
Une associativité à gauche sur un opérateur $N \mapsto NuN$ va casser toutes les paires critiques $(N \mapsto u_1 \underline{N} u N \mid N u \underline{N})$ de la règle avec elle même en interdisant les arbres syntaxiques contenant un sous-arbre de la forme $(N, \boldsymbol{Nu}(N, \boldsymbol{N'u'N''}))$ au profit de l'arbre équivalent $(N, (N, \boldsymbol{NuN'})\boldsymbol{u'N''})$.
Une associativité droite fera de même mais en privilégiant le premier sur le second.

Remarque 8. *Cette définition peut se généraliser à toute règle $N \mapsto w$, mais dans la pratique ce n'est jamais utile.*
En effet, une paire critique entre w et lui même est forcément de la forme $((Nu)^n \underline{N} (uN)^k \mid (Nu)^{n'} \underline{N} (uN)^{k'})$ pour $n + k = n' + k'$ mais $n \neq n'$. On interdit alors tous les arbres syntaxiques contenant un sous-arbre de la forme $(N, \boldsymbol{(Nu)^n}(N, \boldsymbol{(Nu)^{n'} N (Nu)^{k'}})\boldsymbol{(uN)^k})$ au profit de l'arbre équivalent de la forme $(N, \boldsymbol{(Nu)^{n'}}(N, \boldsymbol{(Nu)^n N (Nu)^k})\boldsymbol{(uN)^{k'}})$ lorsque $n' < n$.

2.3 AST

Un AST, pour *Abstract Synctactic Tree*, est ce qui va être rendu en sortie de votre parseur pour être traité par la suite de votre compilateur.

C'est une abstraction de l'arbre syntaxique, cela signifie :

— que l'on peut y supprimer les informations inutiles pour la suite de la compilation, par exemple, les parenthèses ne sont utiles que pour l'analyse syntaxique (parseur) et peuvent donc être enlevées de l'AST,
— que l'on peut modifier la structure de l'arbre pour refléter une grammaire plus aisée, à utiliser, par exemple, les différentes opérations peuvent être remplacées par des noeuds spécifiques,
— que l'on peut ajouter des informations utile pour la suite (ex : dans le projet, on va avoir besoin d'y ajouter la taille attendu du code généré, on peut aussi y ajouter les types).

2.4 Firsts et Follows

Définition 14 (union-epsilon)**.**
Soit $e, f \subseteq \mathcal{T} \uplus \{\epsilon\}$ et , on appelle union-epsilon l'opérateur :

$$e \textcircled{\cup} f := \begin{cases} (e - \{\epsilon\}) \cup f & si\ \epsilon \in e \\ e & sinon \end{cases}$$

correspondant aux élément non-ϵ de e et aux élément de f uniquement ajoutés si $\epsilon \in e$.

Algorithme 1 (firsts)**.**
Donnée : Une grammaire $\mathcal{G} = (\boldsymbol{\mathcal{T}}, \mathcal{N}, S, \tau)$,
Résultat : pour chaque mot $\omega \in (\mathcal{T} \uplus \mathcal{N})^*$ un ensemble $\texttt{first}(\omega) \subseteq \mathcal{T} \uplus \{\epsilon\}$,
Spécification 1 : $\epsilon \in \texttt{first}(\omega)$ ssi $\omega \mapsto^* \epsilon$,

Spécification 2 : $t \in \texttt{first}(\omega)$ *ssi il existe* ω' *tel que* $\omega \mapsto^* t\omega'$

Point fixe :
- *Ensemble :* $\texttt{First} \subseteq \mathcal{N} \times (\mathcal{T} \uplus \{\epsilon\})$
- *Initialisation :* $\texttt{First} := \emptyset$
- *Pas :* $\texttt{First} := \{(N,t) \mid \exists (N \mapsto \omega), t \in \texttt{first}(\omega)\}$

Macros :
$\texttt{first}(\epsilon) := \emptyset$
$\texttt{first}(tu) := \{t\}$
$\texttt{first}(Nu) := \{t \mid (N,t) \in \texttt{First}\} \textcircled{\cup} \texttt{first}(u)$

Algorithme 2 (follows)**.**
Donnée : Une grammaire $\mathcal{G} = (\boldsymbol{\mathcal{T}}, \mathcal{N}, S, \tau)$,
Résultat : pour chaque non-terminal $N \in \mathcal{N}$ *un ensemble* $\texttt{follow}(N) \subseteq \mathcal{T} \uplus \{\$\}$,
Spécification 1 : $\$ \in \texttt{follow}(N)$ *ssi il existe une réduction* $S \mapsto^* \omega N$,
Spécification 2 : $t \in \texttt{follow}(N)$ *ssi il existe une réduction* $M \mapsto^* \omega_1 N t \omega_2$

Point fixe :
- *Ensemble :* $\texttt{Follow} \subseteq \mathcal{N} \times (\mathcal{T} \uplus \{\$\})$
- *Initisation :* $\texttt{Follow} := \{(S, \$)\}$
- *Pas :* $\texttt{Follow} := \{(N,t) \mid \exists (M \mapsto \omega_1 N \omega_2), t \in \texttt{first}(\omega_2) \textcircled{\cup} \texttt{follow}(M)\}$

Macros :
$\texttt{follow}(N) := \{t \mid (N,t) \in \texttt{Follow}\}$

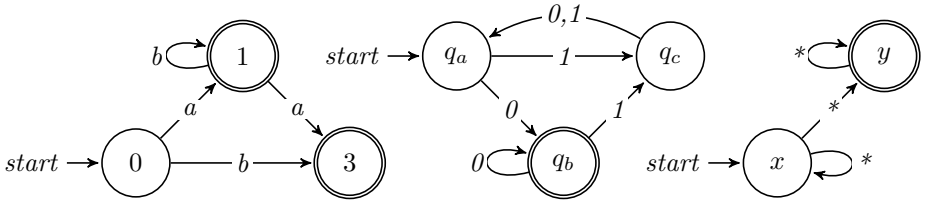
3 Automates : rappels et préliminaires

3.1 Automates Finis

Définition 15 (Automate fini)**.**
Un automate fini $\mathcal{A}$ *est la donnée :*
- *d'un alphabet (de tokens)* $\boldsymbol{\Sigma}$,
- *d'un ensemble d'états* Q,
- *d'un état* $q_0 \in Q$ *initial,*
- *d'un ensemble d'états* $F \subseteq Q$ *finaux,*
- *d'une transition* $\texttt{trans}(q,t) \in Q$ *pour chaque état* $q \in Q$ *et caractère* $t \in \Sigma$.

Notations 6. *On représente un automate fini* $\mathcal{A} = (\boldsymbol{\Sigma}, Q, q_0, F, \texttt{trans})$ *comme un graphe dirigé étiqueté dont les noeuds sont les états* $q \in Q$ *et les arêtes sont les* $p \xrightarrow{t} q$ *tels que* $q = \texttt{trans}(p,t)$*, l'état initial est indiqué par une flèche entrante sans origine, et les états finaux par un double cercle (ou double carré lorsque l'on écrit des états carrés) autour de l'état. Attention,* trans *étant une fonction, il doit y avoir exactement une arête* $p \xrightarrow{t} q$ *pour chaque état* p *et chaque lettre* t*. Pour la lisibilité, on peut fusionner deux arêtes en une seule en plaçant les deux labels côte à côte, séparés par une virgule.*

Exercice 1. *Des graphes ci-dessous, lesquels représentent des automates finis ? Donner leurs définitions formelles*



Définition 16 (reconnaissance sur un automate)**.**
Un mot $\boldsymbol{\omega} \in \boldsymbol{\Sigma}^*$ *sur l'alphabet* $\boldsymbol{\Sigma}$ *est reconnu par un automate fini* $\mathcal{A} = (\boldsymbol{\Sigma}, Q, q_0, F, \texttt{trans})$ *si*
- *ou bien* $\boldsymbol{\omega} = \epsilon$ *est le mot vide et* $q_0 \in F$*,*
- *ou bien* $\boldsymbol{\omega} = \boldsymbol{t}.\boldsymbol{\omega}'$ *et* $\boldsymbol{\omega}'$ *est reconnu par* $(\boldsymbol{\Sigma}, Q, \texttt{trans}(q_0, t), F, \texttt{trans})$*.*

L'ensemble des mots reconnus par $\mathcal{A}$ *est son langage, désigné par* $L_{\mathcal{A}}$

Proposition 4 (reconnaissance)**.**
La complexité en temps de cet algorithme de reconnaissance est

$$O(|\omega| * log(|Q|*|\Sigma|)) \overset{\Sigma}{\simeq}^{cst} O(|\boldsymbol{\omega}| * log(|Q|)).$$

Sa complexité en espace est (en considérant que les tokens sont vides) :

$$O(|\omega| * log(|\Sigma|) + (|Q|*|\Sigma|) * log(Q)) \overset{\Sigma}{\simeq}^{cst} O(|\omega| + |Q|log(|Q|))$$

Démonstration. En temps :
À chaque lettre du mot, on teste si le mot est vide (constant), on récupère la première lettre (constant), on appelle la fonction $\texttt{trans}(q_0, t)$ qui est en $log(|Q| * |\Sigma|)$ dans le cas général. Le fait que la fonction $\texttt{trans}(q_0, t)$ qui est en $log(|Q| * |\Sigma|)$ dans le cas général vient du cas où `trans` est un dictionnaire (associant les clés dans $Q * \Sigma$ à des résultats dans Q) codé avec un arbre équilibré (AVL, ARN, A23, R-tree, etc...), la complexité peut être prouvée optimale. En espace :
Il faut stocker le mot (en $O(|\omega|.log(|\Sigma|)$ si les tokens sont vides, car une lettre prend une taille $log(|\Sigma|)$), un pointeur sur le mot (en $O(log(|\boldsymbol{\omega}|))$) et l'automate lui même. Pour l'automate, on doit stocker (dans le pire cas) le nom d'un état (en $O(log(|Q|))$) pour chaque entrée état/lettre (en $O(|Q|*|\Sigma|)$). □

Remarque 9. *On arrive ici aux limites entre la théorie et la pratique de la complexité : bien que les facteurs logarithmiques soit toujours là, on les ignores généralement en pratique, et à raison ! En effet, en utilisant une structure de tableau, on perd en flexibilité, mais on a un accès qui semble être en temps constant. Je dis bien "semble" car il n'est en temps constant que tant qu'il y a de la place sur votre RAM... ce qui en pratique est toujours vérifié (à moins d'avoir fait explosé votre complexité en espace...).*

Exercice 1 (Automates partiels [2]). Un automate fini partiel est définit de la même façon, sauf que q_0, ainsi que $\texttt{trans}(q,t)$ ne sont pas toujours définis (renvoient $\perp$), lors de la reconnaissance, on renvoi faux lorsque l'on essai de calculer un état non définit.
Montrez que l'on peut réduire tout automate partiel $\mathcal{A}$ en un automate fini $\mathcal{A}$ de même langage. L'augmentation du nombre d'états doit être constante.

Notations 7. *Puisqu'ils sont si proche, on dessinera généralement un automate fini partiel pour parler de l'automate fini. En effet, l'état puits/poubelle a plein de transition qui lui arrive, ce qui complexifie le dessin pour rien.*

Les automates finis ont 5 propriétés remarquables qui justifient leur étude et qui expliquent pourquoi on essai de les étendre :
- la complexité linéaire (en pratique) de la reconnaissance que l'on vient de voir,
- l'équivalence avec les langages rationnels, que l'on va voire tout de suite,
- la minimisation, que l'on va voir la sous section suivante,
- la déterminisation, que l'on va voire la sous section encore après et qui sera le centre de la partie,
- les propriétés de vérifications (langage vide, etc...) que l'on ne verra pas.

3.2 Expressions régulières

3.2.1 Théorie

Définition 17. *Les expressions régulières* $\texttt{Reg}(\Sigma)$ *sur l'alphabet* Σ *sont données par la grammaire suivante :*
- $\mathcal{T} = \Sigma \cup \{(,),^*,+,,\epsilon\}$,
- $\mathcal{N} = \{S\}$,
- *les règles sont :*

$$S \mapsto \boldsymbol{t} \qquad S \mapsto \varepsilon \qquad S \mapsto S + S$$
$$S \mapsto SS \qquad S \mapsto S^* \qquad S \mapsto (S)$$

Ou de façon équivalente (ici `t` *représente tous les symboles de* $t \in \Sigma$*) :*

```
<regexpr> := <CHAR>        | <EPS>                  | <regexpr>+<regexpr>
           | <regexpr>*    | <regexpr><regexpr>     | (<regexpr>)
```

De plus, la somme $S + S$ *et la composition* SS *sont associatif à droite, avec l'étoile* S^* *prioritaire sur la composition elle même prioritaire sur la somme.*

Remarque : Ci-dessus, on utilise le symbole $\varepsilon \neq \epsilon$ pour bien indiquer que la règle ne créé pas le mot vide ϵ, mais crée un symbole ε qui est celui de l'expression régulière vide. [3]

2. Cet exercice est pour ceux qui ne maitrisent pas les automates, ce ne sera pas demander au partiel.
3. Si vous pensez que l'on mixe les niveaux d'abstration, atendez la suite.

Proposition 5. *La grammaire des expression régulière est non ambigüe modulo les règles de priorités et d'associativité,* $\texttt{Reg}(\Sigma)$ *est donc bien définit.*

Notations 8. *On appelle expression régulière à la fois les arbres syntactiques de la grammaire et les mots dérivés. En effet, la grammaire n'étant pas ambigüe il y a une correspondance exacte.*

Définition 18. *Soit* $\boldsymbol{r}$ *une expression régulière sur l'alphabet* Σ.
Le language $\boldsymbol{L_r}$ *reconnu par* $\boldsymbol{r}$ *est définit par induction sur* $\boldsymbol{r}$ *:*
- $\boldsymbol{L_t} = \{\boldsymbol{t}\}$,
- $\boldsymbol{L_\varepsilon} = \boldsymbol{\Sigma}$,
- $\boldsymbol{L_{r_1+r_2}} = \boldsymbol{L_{r_1}} \cup \boldsymbol{L_{r_2}}$,
- $\boldsymbol{L_{r_1 r_2}} = \{\boldsymbol{\omega_1\omega_2} \mid \boldsymbol{\omega_1} \in \boldsymbol{L_{r_1}},\ \boldsymbol{\omega_2} \in \boldsymbol{L_{r_2}}\}$,
- $\boldsymbol{L_{s^*}} = \{\boldsymbol{\omega_1 \ldots \omega_n} \mid n \geq 0, \forall i \leq n, \boldsymbol{\omega_i} \in \boldsymbol{L_s}\}$.

Remarque : La somme et la composition sont commutatives au sens où choisir une associativité à gauche n'aurait pas changé le langage reconnu.

Définition 19. *Soit* r *une expression régulière sur l'alphabet* Σ.
L'automate $\mathcal{A}_r$ *associée à* r *est définit comme la déterminisation et minimisation de l'automate* $\mathcal{A}_r^0 = (\Sigma, Q_r, q_{r,0}, F_r, \texttt{eps}_r, \texttt{trans}_r)$ *définit par induction sur* r *:*
- *cas* t *:*

$$Q_t = \{q_{t,0}, 1\}\,, \qquad F_t = \{1\}\,, \qquad \texttt{eps}_t(q,s) = \emptyset\,,$$

$$\texttt{trans}_t(q,s) \left\{ \begin{array}{cc} \{1\} & si\ q = q_{t,0}\ et\ s = t \\ \emptyset & sinon \end{array} \right\}$$

- *cas* ε *:*

$$Q_\varepsilon = \{q_{\varepsilon,0}, 1\}\,, \qquad F_\varepsilon = \{1\}\,, \qquad \texttt{eps}t.(q,s) = \emptyset\,,$$

$$\texttt{trans}_\varepsilon(q,s) \left\{ \begin{array}{cc} \{1\} & si\ q = q_{\varepsilon,0} \\ \emptyset & sinon \end{array} \right\}$$

- *cas* $r_1 + r_2$ *:*

$$Q_{r_1+r_2} = Q_{r_1} \uplus Q_{r_2} \uplus \{q_{r_1+r_2,0}\}\,, \qquad F_{r_1+r_2} = , F_{r_1} \uplus F_{r_2}$$

$$\texttt{eps}_{r_1+r_2}(q) \left\{ \begin{array}{cc} \{q_{r_1}, q_{r_2}\} & si\ q = q_{r_1+r_2,0} \\ \texttt{eps}_{r_1}(q) & si\ q \in Q_{r_1} \\ \texttt{eps}_{r_2}(q) & si\ q \in Q_{r_2} \end{array} \right\} \qquad \texttt{trans}_{r_1+r_2}(q) \left\{ \begin{array}{cc} \emptyset & si\ q = q_{r_1+r_2,0} \\ \texttt{trans}_{r_1}(q) & si\ q \in Q_{r_1} \\ \texttt{trans}_{r_2}(q) & si\ q \in Q_{r_2} \end{array} \right\}$$

- *cas* $r_1 r_2$ *:*

$$Q_{r_1 r_2} = Q_{r_1} \uplus Q_{r_2}, \qquad q_{r_1 r_2,0} = q_{r_1,0}\,, \qquad F_{r_1 r_2} = , F_{r_2}$$

$$\texttt{eps}_{r_1 r_2}(q) \left\{ \begin{array}{cc} \texttt{eps}_{r_1}(q) & si\ q \in Q_{r_1} - F_{r_1} \\ \texttt{eps}_{r_1}(q) \cup \{q_{r_2,0}\} & si\ q \in F_{r_1} \\ \texttt{eps}_{r_2}(q) & si\ q \in Q_{r_2} \end{array} \right\} \qquad \texttt{trans}_{r_1 r_2}(q) \left\{ \begin{array}{cc} \texttt{trans}_{r_1}(q) & si\ q \in Q_{r_1} \\ \texttt{trans}_{r_2}(q) & si\ q \in Q_{r_2} \end{array} \right\}$$

— *cas* r^* *:*

$$Q_{r^*} = Q_r \uplus \{q_{r^*,0}\}, \qquad F_{r^*} = , F_r$$

$$\texttt{eps}_{r^*}(q) \begin{cases} \{q_{r,0}\} & si\ q = q_{r^*,0} \\ \texttt{eps}_r(q) & si\ q \in Q_{r_1} - F_{r_1} \\ \texttt{eps}_r(q) \cup \{q_{r,0}\} & si\ q \in F_r \end{cases} \qquad \texttt{trans}_{r^*}(q) \begin{cases} \emptyset & si\ q = q_{r^*,0} \\ \texttt{trans}_r(q) & sinon \end{cases}$$

Proposition 6. *Le language reconnu par l'automate associé à une expression régulière est exactement le langage reconnu par celle-ci :*

$$\boldsymbol{L_{\mathcal{A}_r} = L_r}.$$

Remarquez que l'on peut aussi aller dans l'autre sense :

Proposition 7. *Pour tout automate, il existe une expression régulière qui reconnait le même langage*

Celà veut dire que le pouvoir expressif des automates et celui des expressions régulières sont les mêmes, on parle de langages réguliers. Mais ce second sense n'a pas vraiment d'intéret pratique. [4]

3.2.2 Pratique

Dans la pratiques, grammaires et expressions régulières sont intimement liées et définies l'une à partir de l'autre... Nous allons essayer de démêler tout ça.

D'abord, notons qu'en pratique, on utilise principalement les expressions régulières sur des charatères, et que l'on ne veut pas s'ammuser à écrire `'a'+'b'+'c'+...+'z'` lorsque l'on veut représenter toutes les lettres minuscule, on utilise plutôt `'a'-'z'` qui désigne tous les charactères entre 'a' à 'z'.

Définition 20 (Ensembles de charactères)**.** *Le token* `<CHAR>` *sera utiliser pour désigner un charactère (typiquement ASCII, mais on pourait iméginer d'autres encodages) entre deux quotes* `'` *et avec échapements.*
On utilise alors la grammaire suivante pour décrire les ensembles de caracrères :

`<chars>` := `<CHAR>` | `<CHAR>-<CHAR>` | `<CHAR>,<chars>` | `<CHAR>-<CHAR>,<chars>`

Remarques : D'abord, remarquez que l'on ne décrit pas bien ce qu'est un "caractère", normalement, il faudrait l'expression régulière définissant les caractères, qui elle même nécéssite de l'arler d'ensemble de caractères... si on ne veut pas boucler, il faudrait être plus formel, mais ce serait trop...

Aussi utilisé pour les automates : Les ensembles de caractères sont aussi utilisés pour les transition des automates sur des caractères.

4. En tout cas pas en compilation...

Une grammaire plus riche : En pratique, les outils utilisant des expressions régulières utilisent chacun leur grammaire d'expressions régulières, la très grande majorité de ces notations se traduisent avec les opérateurs que nous avons vus, mais certaines sont des "expressions régulières étendues" et ne peuvent s'exprimer que en utilisant une grammaire.

3.3 Minimisation

Définition 21 (Accessibilité et co-accessibilité)**.**
Soit $\mathcal{A} = (\mathbf{\Sigma}, Q, q_0, F, \texttt{trans})$ *automate fini (partiel).*
Un état $q \in Q$ *est dit accessible s'il existe un mot* $t_1 \dots t_n \in \Sigma^n$ *une séquence* $q_0, q_1, \dots, q_n = q$ *tels que* $q_{i+1} = \texttt{trans}(q_i, t_{i+1})$*, c'est à dire que* q *peut être accédé depuis l'état initial* q_0*.*
Un état $q \in Q$ *est dit co-accessible s'il existe un mot* $t_1 \dots t_n \in \Sigma^n$ *une séquence* $q = p_0, p_1, \dots, p_n \in F$ *tels que* $p_{i+1} = \texttt{trans}(p_i, t_{i+1})$*, c'est à dire qu'un état final* $p_n \in F$ *peut être accédé depuis* q*.*

Algorithme 3 (Co-accessibilité)**.**
Donnée : Un automate fini $\mathcal{A} = (\mathbf{\Sigma}, Q, q_0, F, \texttt{trans})$*,*
Résultat : un automate fini partiel $\texttt{cac}(\mathcal{A}) = (\mathbf{\Sigma}, \texttt{cac}(Q), q_0', F, \texttt{trans})$*,*
Spécification : $L_{\mathcal{A}} = L_{\texttt{cac}(\mathcal{A})}$ *et* $\texttt{cac}(Q) \subseteq Q$ *et tous les états de* $\texttt{cac}(\mathcal{A})$ *co-accessibles.*

Point fixe :
- *Ensemble :* $\texttt{cac}(Q) \subseteq Q$
- *Initialisation :* $\texttt{cac}(Q) := F$*,*
- *Pas :* $\texttt{cac}(Q) := \texttt{cac}(Q) \cup \{q \mid \exists t \in \Sigma, \texttt{trans}(q, t) \in \texttt{cac}(Q)\}$*.*

À la fin, on doit vérifier que $q_0 \in \texttt{cac}(Q)$*, si c'est le cas* $q_0' = q_0$ *sinon* $q_0' = \bot$ *est non définit.*

Algorithme 4 (Accessibilité)**.**
Donnée : Un automate fini partiel $\mathcal{A} = (\mathbf{\Sigma}, Q, q_0, F, \texttt{trans})$*,*
Résultat : un automate fini partiel $\texttt{acc}(\mathcal{A}) = (\mathbf{\Sigma}, \texttt{acc}(Q), q_0, F', \texttt{trans})$
Spécification : $L_{\mathcal{A}} = L_{\texttt{acc}(\mathcal{A})}$ *et* $\texttt{acc}(Q) \subseteq Q$ *et tous les états de* $\texttt{acc}(\mathcal{A})$ *accessibles.*

Point fixe :
- *Ensemble :* $\texttt{acc}(Q) \subseteq Q$
- *Initialisation :* $\texttt{acc}(Q) := \{q_0\}$ *si* $q_0 \neq \bot$ *et* $\emptyset$ *sinon,*
- *Pas :* $\texttt{acc}(Q) := \texttt{acc}(Q) \cup \{\texttt{trans}(q, t) \mid q \in \texttt{acc}(Q), t \in \Sigma\}$*.*

À la fin on pose $F' = F \cap \texttt{acc}(Q)$*.*

Algorithme 5 (Minimisation)**.**
Donnée : Un automate fini $\mathcal{A} = (\mathbf{\Sigma}, Q, q_0, F, \texttt{trans})$*,*
Résultat : un automate fini $\texttt{min}(\mathcal{A}) = (\mathbf{\Sigma}, \texttt{min}(Q), q_0^m, F^m, \texttt{trans}^m)$ *ou une erreur,*
Spécification : $L_{\mathcal{A}} = L_{\texttt{min}(\mathcal{A})}$ *et* $\texttt{min}(\mathcal{A})$ *est minimal parmi les automates ayant cette propriété.*

Étape 1 : calculer $\texttt{cac}(\mathcal{A})$ *avec l'Algorithme 3,*
Étape 2 : calculer $\texttt{acc}(\texttt{cac}(\mathcal{A}))$ *avec l'Algorithme 4,*
Étape 3 : rajouter un état puit pour retrouver un automate total $\mathcal{A}' = (\mathbf{\Sigma}, Q', q_0', F', \texttt{trans})$
Étape 4 : on cherche une relation d'équivalence $(\equiv)$ *sur* Q' *appelée bissimulation, pour ça on procède par point fixe :*

- *Ensemble :* $(\equiv) \subseteq {Q'}^2$,
- *Initialisation :* $(\equiv) := (=)$ *c'est à dire que deux états sont équivalents ssi ils sont égaux,*
- *Pas :* $p \equiv q$ *lorsque :*
  - $(p \in F \Leftrightarrow q \in F)$,
  - *et pour tout* $t \in \Sigma$, $\texttt{trans}(p,t) \equiv \texttt{trans}(q,t)$.

Étape 5 : On pose alors

- $\texttt{min}(Q) := Q_{/\equiv}$ *est l'ensemble des classes d'équivalences de* $\equiv$,
- $q_0^m := [q_0]$ *est la classe d'équivalence de* q_0,
- $F^m := F_{/\equiv}$ *est l'ensemble des classes d'équivalences de* $\equiv$ *sur* F,
- $\texttt{trans}^m([q],t) := [\texttt{trans}(q,t)]$

3.4 Automates non déterministes et déterminisation

Définition 22 (Automate fini non déterministe)**.**
Un automate fini non déterministe $\mathcal{A}$ *est la donnée :*

- *d'un alphabet (de tokens)* $\mathbf{\Sigma}$,
- *d'un ensemble d'états* Q,
- *d'un état* $q_0 \in Q$ *initial,*
- *d'un ensemble d'état* $F \subseteq Q$ *finaux,*
- *d'un ensemble d'*ϵ*-transitions* $\texttt{eps}(q) \subseteq Q$ *pour chaque état* $q \in Q$,
- *d'un ensemble de transitions* $\texttt{trans}(q,t) \subseteq Q$ *pour chaque état* $q \in Q$ *et caractère* $t \in \Sigma$.

Notations 9. *On représente un automate fini non déterministe* $\mathcal{A} = (\mathbf{\Sigma}, Q, q_0, F, \texttt{eps}, \texttt{trans})$ *comme un graphe dirigé étiqueté dont les noeuds sont les états* $q \in Q$ *et les arrêtes sont les* $p \xrightarrow{t} q$ *tels que* $q \in \texttt{trans}(p,t)$ *et les* $p \xrightarrow{\epsilon} q$ *tels que* $q \in \texttt{eps}(p)$.

Définition 23 (reconnaissance sur un automate non déterministe)**.**
Un mot $\boldsymbol{\omega} \in \mathbf{\Sigma}^*$ *sur l'alphabet* $\mathbf{\Sigma}$ *est reconnu par un automate fini non déterministe* $\mathcal{A} = (\mathbf{\Sigma}, Q, q_0, F, \texttt{eps}, \texttt{trans})$ *si*

- *ou bien* $\boldsymbol{\omega} = \epsilon$ *est le mot vide et* $q_0 \in F$,
- *ou bien* $\boldsymbol{\omega}$ *est reconnu par* $(\mathbf{\Sigma}, Q, q', F, \texttt{trans})$ *pour un état* $q' \in \texttt{eps}(q)$,
- *ou bien* $\boldsymbol{\omega} = \boldsymbol{t}.\boldsymbol{\omega}'$ *et* $\boldsymbol{\omega}'$ *est reconnu par* $(\mathbf{\Sigma}, Q, q', F, \texttt{trans})$ *pour un état* $q' \in \texttt{trans}(q_0, t)$.

L'ensemble des mots reconnus par $\mathcal{A}$ *est son langage, désigné par* $L_{\mathcal{A}}$

Proposition 8 (reconnaissance non déterministe)**.**
Cet algorithme naïf de reconnaissance est en

$$|Q|^{O(|\omega|*|Q|)} * log(|\Sigma|)^{O(|\omega|)} \overset{\Sigma}{\simeq}^{cst} |Q|^{O(|\omega|*|Q|)}.$$

à retenir : plus que exponentiel en $|Q|$ *et en* $|\omega|$

Démonstration. Pour éviter de boucler, on remarque que l'on ne peut pas prendre une ϵ-transition plus de $|Q|$ fois de suite.
Soit $f_k(\omega)$ le temps de reconnaissance du mot ω depuis un état quelconque alors que l'on vient de prendre k ϵ-transitions de suite.
On a alors

$$\begin{aligned} f_{|Q|}(\epsilon) &= O\left(log(|Q|)\right) \\ f_k(\epsilon) &= O\left(log(|Q|) + log(|Q|) * |Q| * f_{k+1}(\epsilon)\right) \\ f_{|Q|}(t\omega) &= O\left(|Q| * log(Q*|\Sigma|) * f_0(\omega)\right) \\ f_k(t\omega) &= O\left(|Q| * log(Q*|\Sigma|) * f_0(\omega) + log(|Q|) * |Q| * f_{k+1}(\epsilon)\right) \end{aligned}$$

On a donc :

$$\begin{aligned} f_k(\epsilon) &= O\left(log(|Q|) * \frac{(|Q| * log(|Q|))^{|Q|-k+1} - 1}{|Q| * log(|Q|) - 1}\right) \\ &= O\left(|Q|^{|Q|+1-k}\right) \\ f_k(t\omega) &= O\left(log(|Q|*|\Sigma|) * |Q| * \frac{(|Q| * log(|Q|))^{|Q|-k+1} - 1}{|Q| * log(|Q|) - 1} * f_0(\omega)\right) \\ &= O\left(|Q|^{|Q|+2-k} log(|\Sigma|) * f_0(\omega)\right) \end{aligned}$$

soit :

$$f_0(\omega) = O\left(|Q|^{(|Q|+2)*(|\omega|+1)-1} log(|\Sigma|)^{|\omega|}\right) = |Q|^{O(|\omega|*|Q|)} * log(|\Sigma|)^{O(|\omega|)}$$

□

Theorem 1 (déterminisation)**.**
Tout automate fini non déterministe se réduit à un automate fini de même langage. Le nombre d'états de l'automate créé est au plus $2^{|Q|}$ où $|Q|$ est le nombre d'états de celui de départ.

Démonstration. On commence éliminer les ϵ-transitions, càd. par réduire notre automate fini non déterministe en un automate fini non déterministe sans ϵ-transitions avec le même nombre d'état.
On nomme cet automate non déterministe sans ϵ-transitions $\mathcal{A} = (\mathbf{\Sigma}, Q, I, F, \texttt{trans})$.
On utilise alors l'automate fini $\texttt{det}(\mathcal{A}) := (\mathbf{\Sigma}, Q^*, I, F^*, \texttt{trans}^*)$ où :

- Q^* est l'ensemble $\mathcal{P}(Q)$ des sous-ensembles de Q,
- $I \in \mathcal{P}(Q)$ est bien un état de $\texttt{det}(\mathcal{A})$,
- $F^* := \{A \subseteq Q \mid A \cap F \neq \emptyset\}$ est l'ensemble des ensembles d'états contenant un terminal,
- $\texttt{trans}^*(A, t) := \bigcup_{q \in A} \texttt{trans}(q, t)$ est l'ensemble des états accessibles depuis l'un des états de A en prenant une t-transition.

On prouve par induction sur $|\omega|$ que tout $\boldsymbol{\omega} \in \boldsymbol{\Sigma}^*$ est reconnu par $\mathcal{A}$ si et seulement s'il est reconnu par $\mathtt{det}(\mathcal{A})$:

— ϵ est reconnu par $\mathcal{A}$ ssi $I \cap F \neq \emptyset$, c'est à dire ssi $I \in F^*$,

— $\boldsymbol{t}.\boldsymbol{\omega}'$ est reconnu par $\mathcal{A}$ ssi $\boldsymbol{\omega}'$ est reconnu par $(\boldsymbol{\Sigma}, Q, \bigcup_{q \in I} \mathtt{trans}(q,t), F, \mathtt{trans}) = (\boldsymbol{\Sigma}, Q, \mathtt{trans}^*(I,t), F, \mathtt{trans})$; par hypothèse d'induction sur $|\omega'| < |\omega|$, cela est équivalent à ω' reconnu par $(\boldsymbol{\Sigma}, Q^*, \mathtt{trans}^*(I,t), F^*, \mathtt{trans}^*)$, or cela signifie exactement que ω est reconnu par $\mathtt{det}(\mathcal{A})$.

□

Algorithme 6 (déterminisation)**.**
Donnée : Un automate fini non déterministe $\mathcal{A} = (\boldsymbol{\Sigma}, Q, q_0, F, \mathtt{eps}, \mathtt{trans})$,
Résultat : un automate fini déterministe $\mathtt{det}(\mathcal{A}) = (\boldsymbol{\Sigma}, Q_*, q_*, F_*, \mathtt{trans}^*)$
Spécification : $L_{\mathcal{A}} = L_{\mathtt{det}(\mathcal{A})}$ *et* $|Q'| \leq 2^{|Q|}$.

Point fixe :

— *Ensemble :* $Q_* \subseteq Q^* := \mathcal{P}(Q)$,

— *Initialisation :* $Q_* := \{\mathtt{eps}^*(\{q_0\})\}$,

— *Pas :* $Q_* := Q_* \cup \{\mathtt{eps}^*(\mathtt{trans}^*(A,t)) \mid A \in Q_*, t \in \Sigma\}$.

Sous fonctions :

— $\mathtt{eps}^*(A) \subseteq Q$ *est la clôture de* $A \subseteq Q$ *par* eps *comme définit Exemple 2.*

— $\mathtt{trans}^*(A,t) := \bigcup_{q \in A} \mathtt{trans}(q,t)$ *pour* $A \subseteq Q$ *et* $t \in \Sigma$.

Proposition 9. *Une fois déterminisé, la reconnaissance d'un mot* $\boldsymbol{\omega}$ *par un automate fini non déterministe* $\mathcal{A} = (\boldsymbol{\Sigma}, Q, q_0, F, \mathtt{eps}, \mathtt{trans})$ *est en :*

$$O\left(|\omega| * (|Q| + log|\Sigma|)\right) \overset{\Sigma}{\simeq}^{cst} O\left(|\omega| * |Q|\right) \ .$$

Remarque 10. *L'ensemble* Q_* *d'états trouvés est inférieur à la borne* Q^* *donnée dans la preuve du théorème :*

$$Q_* \subseteq Q^* \ .$$

Bien qu'en théorie, elle peut atteindre cette borne, en pratique elle est TRÈS inférieure : souvent polynomial en $|Q|$ *et non exponentielle ! Cela signifie, qu'en pratique, la reconnaissance après déterminisation est souvent en*

$$O\left(|\omega| * log(|Q| * |\Sigma|)\right) \overset{\Sigma}{\simeq}^{cst} O\left(|\omega| * log(|Q|)\right) \ .$$

4 Automates shift-action

Intuition : Un automate shift-action resemble a un automate habituel, sauf qu'il n'a pas d'états finaux, mais des transitions spécifiques, appellées actions, qui vont lancer une sous-routine. Celles-ci peuvent avoir lieux au millieu du mot (si on lit un charactère que l'on ne comprends pas par exemple), ou à la fin (comme l'acceptation sur l'automate habituel). On utilisera le symbole \$ pour désigner la fin de mot (correspond à EOF ou `end_of_file` dans les outils commerciaux).

Définition 24 (Automate shift-action)**.**
Un automate shift-action $\mathcal{A}$ est la donnée :
- *d'un alphabet (de tokens) $\boldsymbol{\Sigma}$,*
- *d'un ensemble d'actions possibles Δ avec le code associé dont une spécifique* `Err` *qui échoue,*
- *d'un ensemble d'états Q,*
- *d'un état $q_0 \in Q$ initial,*
- *d'un choix d'action $C(q) \subseteq \Sigma$ pour chaque état $q \in Q$,*
- *d'une transition (ou shift) $\texttt{shift}(q,t) \in Q$ pour chaque état $q \in Q$ et caractère $t \in \Sigma - C(q)$,*
- *d'une action $\texttt{action}(q,t) \in \Delta$ pour chaque état $q \in Q$ et caractère $t \in C(q) \cup \{\$\}$.*

Les actions $\delta \in \Delta$ sont des codes code arbitraires qui dépendent de deux paramètres : le buffer $\boldsymbol{u} \in \boldsymbol{\Sigma}^*$ et le mot qui restera à lire $\boldsymbol{\omega} \in \boldsymbol{\Sigma}^*$.

Exemple 4.

1. *L'automate shift-action $\mathcal{A}_\perp = (\boldsymbol{\Sigma}, \texttt{Err}, \{\perp\}, \perp, C, \texttt{shift}, \texttt{action})$ où :*
   - *C définit par $C(\perp) = \Sigma$*
   - *on n'a pas de shift car $\Sigma - C(\perp) := \emptyset$,*
   - *on renvoi toujours une erreur avec $\texttt{action}(\perp, t) := \texttt{Err}$ pour tout t.*
2. *Un automate $\mathcal{A}$ est un automate shift-action si on considère $(\boldsymbol{\Sigma}, \{\texttt{Acc}, \texttt{Err}\}, \{\perp\}, \perp, C, \texttt{shift}, \texttt{action})$ où :*
   - `Acc` *se contente d'accepter le mot,*
   - *C définit par $C(a) = \emptyset$,*
   - *$\texttt{shift}(q,a) = \delta_A(q,a)$,*
   - *$\texttt{action}(q,\$) := \texttt{Acc}$ si $q \in F_A$ et $\texttt{action}(q,\$) := \texttt{Err}$ sinon.*
3. *Un automate $\mathcal{A}$ avec puits positifs et négatifs est un automate shift-action $(\boldsymbol{\Sigma}, \{\texttt{Acc}, \texttt{Err}\}, \{\perp\}, \perp, C, \texttt{shift}, \texttt{action})$ où :*
   - `Acc` *se contente d'accepter le mot,*
   - *C définit par $C(a)$ est l'ensemble des transitions vers un puit (positif ou négatif),*
   - *$\texttt{shift}(q,a) = \delta_A(q,a)$ sont les transitions n'allant pas vers des puits,*
   - *$\texttt{action}(q,\$) := \texttt{Acc}$ si $q \in F_A$ et $\texttt{action}(q,\$) := \texttt{Err}$ sinon,*
   - *$\texttt{action}(q,a) := \texttt{Acc}$ si la transition $\delta(q,a)$ est un puits positif et $\texttt{action}(q,a) := \texttt{Err}$ si c'est un puits négatif.*

Notations 10. *Pour des raisons de lisibilité, on ne dessinera généralement pas les actions* `Err`*. De plus, il y a généralement peu d'états avec des actions (autres que* `Err`*), on notera ces états avec un double cercle (ou double carré lorsque l'on écrit des états carrés) et les action qui en partent avec des flèches étiquetées sortantes ; on peut se permettre de ne pas marquer les actions sortantes lorsqu'elles sont décrites par ailleurs.*

Définition 25 (Exécution sur un automate shift-action)**.**
On considère un automate shift-action $\mathcal{A} = (\boldsymbol{\Sigma}, \Delta, Q, q_0, C, \texttt{shift}, \texttt{action})$, un mot ω et un buffer $u \in \boldsymbol{\Sigma}^*$ (vide au départ). L'exécution de $\boldsymbol{\omega}$ sur $\mathcal{A}$ se déroule par induction :

— *argument inductif :* $\boldsymbol{\omega}$,
— *Cas* ϵ *: on exécute l'action* $\texttt{action}(q_0, \$)(\boldsymbol{u}, \epsilon)$,
— *Cas* $\boldsymbol{t\omega}$ *lorsque* $t \in C(q)$ *: on exécute l'action* $\texttt{action}(q_0, t)(\boldsymbol{u}, \boldsymbol{\omega})$,
— *Cas* $\boldsymbol{t\omega}$ *lorsque* $t \notin C(q)$ *: on place* $\boldsymbol{t}$ *dans le buffer* $\boldsymbol{u}$*, et on exécute l'automate shift-action* $(\boldsymbol{\Sigma}, Q, \Delta, \texttt{shift}(q_0, t), C, \texttt{shift}, \texttt{action})$ *sur* $\boldsymbol{\omega}$.

On note $L_{\mathcal{A}}$ *l'ensemble des quadruplets* $(\boldsymbol{\omega}, \delta, \boldsymbol{\omega}', \boldsymbol{u}) \in \boldsymbol{\Sigma}^* \times \Delta \times \boldsymbol{\Sigma}^* \times \boldsymbol{\Sigma}^*$ *telle que* $\mathcal{A}$ *exécuté sur* $\boldsymbol{\omega}$ *va lancer* δ *sur le buffer* $\boldsymbol{u}$ *et le résidu* $\boldsymbol{\omega}'$.

Remarque 11. *Si* $(\boldsymbol{\omega}, \delta, \boldsymbol{\omega}', u) \in L_{\mathcal{A}}$*, alors* $\boldsymbol{\omega} = u.\boldsymbol{\omega}'$.

Remarque 12. *Ce formalisme est très (trop ?) lax, au sens où une action est capable de faire n'importe quoi, y compris relancer l'automate ou utiliser le buffer u (c'est d'ailleurs pour ça qu'il est là). Ce qui est important est que la phase avec les shifts se comporte comme un automate fini, on peut donc la minimiser, ainsi que déterminiser sa version non déterministe !*

Algorithme 7 (Co-acessibilité)**.**
Donnée : Un automate shift-action $\mathcal{A} = (\boldsymbol{\Sigma}, \Delta, Q, q_0, C, \texttt{shift}, \texttt{action})$,
Résultat : un automate shift-action $\texttt{cac}(\mathcal{A}) = (\boldsymbol{\Sigma}, \Delta, \texttt{cac}(Q), q_0', C', \texttt{shift}, \texttt{action}')$,
Spécification : $L_{\mathcal{A}} = L_{\texttt{cac}(\mathcal{A})}$ *et* $\texttt{cac}(Q) \subseteq Q$ *et tous les états de* $\texttt{cac}(\mathcal{A})$ *co-accessibles.*

Point fixe :
— *Ensemble :* $\texttt{cac}(Q) \subseteq Q$
— *Initialisation :* $\texttt{cac}(Q) := \{q \mid \exists t \in C(q) \cup \{\$\}, \texttt{action}(q, t) \neq \texttt{Err}\}$,
— *Pas :* $\texttt{cac}(Q) := \texttt{cac}(Q) \cup \{q \mid \exists t \in \Sigma - C(q), \texttt{trans}(q, t) \in \texttt{cac}(Q)\}$.

À la fin, on doit vérifier que $q_0 \in \texttt{cac}(Q)$*, si c'est le cas* $q_0' = q_0$ *sinon on rend l'automate shift-action qui rend toujours une erreur* $\texttt{cac}(\mathcal{A}) := \mathcal{A}_\perp$*. De plus, on complète les transitions libres avec des erreurs :*
$F'(q) := \{t \mid t \in C(q)$ *où* $\texttt{shift}(q, t) \notin Q_*\}$,

$$\texttt{action}'(q, t) := \begin{cases} \texttt{action}(q, t) & \textit{si} \quad t \in C(q) \\ \texttt{Err} & \textit{si} \quad \texttt{shift}(q, t) \notin Q_* \end{cases}$$

Algorithme 8 (Acessibilité)**.**
Donnée : Un automate shift-action $\mathcal{A} = (\boldsymbol{\Sigma}, \Delta, Q, q_0, F, \texttt{shift}, \texttt{action})$,
Résultat : un automate shift-action partiel $\texttt{acc}(\mathcal{A}) = (\boldsymbol{\Sigma}, \Delta, \texttt{acc}(Q), q_0, F, \texttt{shift}, \texttt{action})$
Spécification : $L_{\mathcal{A}} = L_{\texttt{acc}(\mathcal{A})}$ *et* $\texttt{acc}(Q) \subseteq Q$ *et tous les états de* $\texttt{acc}(\mathcal{A})$ *accessibles.*

Point fixe :
— *Ensemble :* $\texttt{acc}(Q) \subseteq Q$
— *Initialisation :* $\texttt{acc}(Q) := \{q_0\}$ *si* $q_0 \neq \perp$ *et* $\emptyset$ *sinon,*
— *Pas :* $\texttt{acc}(Q) := \texttt{acc}(Q) \cup \{\texttt{shift}(q, t) \mid q \in \texttt{acc}(Q), t \in \Sigma - C(q)\}$.

Algorithme 9 (Minimisation)**.**
Donnée : Un automate shift-action $\mathcal{A} = (\boldsymbol{\Sigma}, \Delta, Q, q_0, F, \texttt{shift}, \texttt{action})$,
Résultat : un automate shift-action $\texttt{min}(\mathcal{A}) = (\boldsymbol{\Sigma}, \Delta, \texttt{min}(Q), q_0^m, F^m, \texttt{shift}^m, \texttt{action}^m)$
Spécification : $L_{\mathcal{A}} = L_{\texttt{min}(\mathcal{A})}$ *et* $\texttt{min}(\mathcal{A})$ *est minimal parmi les automates ayant cette propriété.*

Étape 1 : calculer $\texttt{cac}(\mathcal{A})$ *avec l'Algorithme 7,*
Étape 2 : calculer $\texttt{acc}(\texttt{cac}(\mathcal{A}))$ *avec l'Algorithme 8,*
Étape 3 : rajouter un état puits et une erreur pour retrouver un automate total $\mathcal{A}' = (\mathbf{\Sigma}, Q', q_0', F', \texttt{trans})$
Étape 4 : on cherche une relation d'équivalence $(\equiv)$ *sur* Q' *appelée bissimulation, pour ça on procède par point fixe :*

- *Ensemble :* $(\equiv) \subseteq {Q'}^2$,
- *Initialisation :* $(\equiv) := (=)$ *c'est à dire que deux états sont équivalents ssi ils sont égaux,*
- *Pas :* $p \equiv q$ *lorsque :*
  - $C(p) = C(q)$,
  - *pour tout* $t \in \Sigma - C(p)$, $\texttt{shift}(p,t) \equiv \texttt{shift}(q,t)$,
  - *et pour tout* $t \in C(p) \cup \{\$\}$, $\texttt{action}(p,t) = \texttt{action}(q,t)$.

Étape 5 : On pose alors

- $\texttt{min}(Q) := Q_{/\equiv}$ *est l'ensemble des classes d'équivalences de* $\equiv$,
- $q_0^m := [q_0]$ *est la classe d'équivalence de* q_0,
- $F^m([p]) := [C(p)]$,
- $\texttt{shift}^m([q], t) := [\texttt{shift}(q,t)]$
- $\texttt{action}^m([q], t) := \texttt{action}(q,t)$

Définition 26 (Automate shift-action non déterministe)**.**
Un automate shift-action non déterministe $\mathcal{A}$ *est la donnée :*

- *d'un alphabet (de tokens)* $\mathbf{\Sigma}$,
- *d'un ensemble d'états* Q,
- *d'un ensemble d'actions possibles* Δ *avec le code associé,*
- *d'un état* $q_0 \in Q$ *initial,*
- *d'un ensemble d'*ϵ*-transitions* $\texttt{eps}(q) \subseteq Q$ *pour chaque état* $q \in Q$,
- *d'un ensemble de transitions* $\texttt{shift}(q,t) \subseteq Q$ *pour chaque état* $q \in Q$ *et caractère* $t \in \Sigma$
- *d'un ensemble d'actions* $\texttt{action}(q,t) \subseteq \Delta$ *pour chaque état* $q \in Q$ *et caractère* $t \in \Sigma \cup \$$.

Définition 27 (Automate shift-action psedo-déterministe)**.**
Un automate shift-action psedo-déterministe $\mathcal{A}$ *est la donnée :*

- *d'un alphabet (de tokens)* $\mathbf{\Sigma}$,
- *d'un ensemble d'actions possibles* Δ,
- *d'un ensemble d'états* Q,
- *d'un état* $q_0 \in Q$ *initial,*
- *d'un choix* $C(q) \subseteq \Sigma$ *pour chaque état* $q \in Q$,
- *d'une transition* $\texttt{trans}(q,t) \in Q$ *pour chaque état* $q \in Q$ *et caractère* $t \in \Sigma - C(q)$
- *d'un ensemble d'actions* $\texttt{action}(q,t) \subseteq \Delta$ *pour chaque état* $q \in Q$ *et caractère* $t \in \Sigma \cup \$$.

Algorithme 10 (psedo-déterminisation)**.**
Donnée : Un automate shift-action non déterministe $\mathcal{A} = (\mathbf{\Sigma}, Q, \Delta, q_0, \texttt{eps}, \texttt{shift}, \texttt{action})$,
Résultat : un automate shift-action psedo-déterministe $\texttt{det}(\mathcal{A}) = (\mathbf{\Sigma}, \Delta \cup \{\texttt{Err}\}, Q_*, q_0^*, C, \texttt{shift}^*, \texttt{action}^*)$

Spécification : $L_{\mathcal{A}} = L_{\mathtt{det}(\mathcal{A})}$ *et* $|Q'| \leq 2^{|Q|}$.

Point fixe :
- *Ensemble :* $Q_* \subseteq \mathcal{P}(Q) - \{\emptyset\}$ *les ensembles non vides d'états,*
- *Initialisation :* $Q_* := \{\mathtt{eps}^*(\{q_0\})\}$,
- *Pas :* $Q_* := Q_* \cup \{\mathtt{eps}^*(\mathtt{shift}^*(A,t)) \mid A \in Q_*, t \in \Sigma\}$.

Sous fonctions :
- $\mathtt{eps}^*(A) \subseteq Q$ *est la clôture de* $A \subseteq Q$ *par* `eps` *comme définit Exemple 2,*
- $C(A) := \{t \mid \mathtt{shift}^*(A,t) = \emptyset\}$,
- $\mathtt{shift}^*(A,t) := \bigcup_{q\in A} \mathtt{shift}(q,t)$ *pour* $A \subseteq Q$ *et* $t \notin C(A)$,
- $\mathtt{action}^*(A,t) := \bigcup_{q\in A} \mathtt{action}(q,t)$ *pour* $A \subseteq Q$ *et* $t \notin C(A)$,
- $\mathtt{action}^*(A,t) := \bigcup_{q\in A} \mathtt{action}(q,t)$ *pour* $A \subseteq Q$ *et* $t \in C(A)$ *si* $\bigcup_{q\in A} \mathtt{action}(q,t)$ *est non vide,*
- $\mathtt{action}^*(A,t) := \{\mathtt{Err}\}$ *pour* $A \subseteq Q$ *et* $t \in C(A)$ *si* $\bigcup_{q\in A} \mathtt{action}(q,t)$ *est vide,*

Proposition 10 (déterminisation des automates shift-action)**.**
Soit $\mathtt{det}(\mathcal{A}) = (\mathbf{\Sigma}, Q, \Delta, q_0, \mathtt{shift}, \mathtt{action})$ *un automate psedo-déterministe. Cet automate est déterministe si pour tout* $q \in Q$ *et tout* $t \in \Sigma$ *:*

1. *si* $t \in C(q)$, $\mathtt{action}(q,t)$ *est un singleton,*
2. *si* $t \notin C(q)$, $\mathtt{action}(q,t)$ *est un vide.*

Définition 28 (conflits shift/action et action/action)**.**
Soit $\mathtt{det}(\mathcal{A}) = (\mathbf{\Sigma}, Q, \Delta, q_0, \bot, \mathtt{shift}, \mathtt{action})$ *un automate psedo-déterministe. On dit qu'il y a*

1. *un conflit action/action s'il existe* $q \in Q$ *et* $t \in \Sigma$ *tel que* $|\mathtt{action}(q,t)|>1$,
2. *un conflit shift/action s'il existe* $q \in Q$ *et* $t \in \Sigma$ *tel que* $|\mathtt{action}(q,t)| \geq 1$ *et* $\mathtt{shift}(q,t) \neq \bot$.

Définition 29 (Automate shift-action avec priorité)**.**
Un automate shift-action avec priorité est un automate shift-action non déterministe $\mathcal{A} = (\mathbf{\Sigma}, Q, \Delta, q_0, \mathtt{eps}, \mathtt{shift}, \mathtt{action})$ *avec une notion de priorité sur les actions. Celle-ci prend la forme d'un ordre stricte* $(<)$ *sur* Δ.

Proposition 11 (déterminisation avec priorité)**.**
Donnée : Un automate shift-action avec priorité $\mathcal{A} = (\mathbf{\Sigma}, Q, (\Delta, <), q_0, \mathtt{eps}, \mathtt{shift}, \mathtt{action})$,
Résultat : un automate shift-action $\mathtt{det}(\mathcal{A}) = (\mathbf{\Sigma}, Q_*, \Delta, q_0^*, C, \mathtt{shift}^*, \mathtt{action}^*)$
Spécification : $L_{\mathcal{A}} = L_{\mathtt{det}(\mathcal{A})}$ *et* $|Q'| \leq 2^{|Q|}$.

Point fixe :
- *Ensemble :* $Q_* \subseteq Q^* := \mathcal{P}(Q) - \{\emptyset\}$,
- *Initialisation :* $Q_* := \{\mathtt{eps}^*(\{q_0\})\}$,
- *Pas :* $Q_* := Q_* \cup \{\mathtt{eps}^*(\mathtt{shift}^*(A,t)) \mid A \in Q_*, t \in \Sigma\}$.

Sous fonctions :
- $\mathtt{eps}^*(A) \subseteq Q$ *est la clôture de* $A \subseteq Q$ *par* `eps` *comme définit Exemple 2.*
- $C(A) := \{t \mid \exists q \in A, \mathtt{shift}(q,t) \neq \emptyset\}$,

— $\texttt{shift}^*(A,t) := \bigcup_{q\in A} \texttt{shift}(q,t) \in \mathcal{P}(Q) - \{\emptyset\}$ *pour* $A \subseteq Q$ *et* $t \in \Sigma - C(A)$,
— $\texttt{action}^*(A,t) := \bigwedge \texttt{action}(q,t)$ *pour* $A \subseteq Q$ *et* $t \in C(A) \cup \{\$\}$ *, est l'action la plus basse dans l'ordre de priorité, avec* $\bigwedge \emptyset = \texttt{Err}$.

5 Analyse lexicale

5.1 Lexeur

Un fonction de lexing est un programme qui prend en entrée une chaîne de caractères et rend en sortie un mot de token. Ici (pour plus d'uniformité) on la généralise comme un programme qui prend en entrée un mot de token et rend en sortie un autre mot de token.

Définition 30 (Schéma de lexing)**.**
Un schéma de lexing est la donnée :
— *d'un alphabet (de tokens)* $\boldsymbol{\Sigma}$ *d'entrée,*
— *d'un alphabet (de tokens)* $\boldsymbol{\Gamma}$ *de sortie,*
— *d'une séquence* $r_1, ..., r_n \in \texttt{Reg}(\Sigma)$ *d'expressions régulières sur* Σ,
— *pour tout* $i \leq n$, *d'une fonction* $\delta_i(\boldsymbol{\omega}) \in \boldsymbol{\Gamma}$ *qui à chaque mot (de token)* $\boldsymbol{\omega} \in \boldsymbol{L_{r_i}}$ *reconnu par le l'expression régulière* r_i, *associe un token de* $\boldsymbol{\Gamma}$

Définition 31 (Fonction de lexing)**.**
Une fonction de lexing $\mathcal{L} : \boldsymbol{\Sigma}^* \to \boldsymbol{\Gamma}^* \cup \{\texttt{Err}\}$ *est correct pour un schéma de lexing* $(\boldsymbol{\Sigma}, \boldsymbol{\Gamma}, (r_i)_{i\leq n}, (\delta_i)_{i\leq n})$ *si pour tout* $\boldsymbol{\omega} \in \boldsymbol{\Sigma}^*$ *:*
— *ou bien* $\mathcal{L}(\boldsymbol{\omega}) = \texttt{Err}$,
— *ou bien* $\boldsymbol{\omega} = \boldsymbol{\omega_1 ... \omega_k}$ *et il exists* $i_1, ..., i_k \in [1, n]$ *tels que pour tout* $j \leq k$, $\omega_j \in L_{r_{i_j}}$ *est reconnu par* r_{i_j} *et* $\mathcal{L}(\boldsymbol{\omega}) = \delta_{i_1}(\boldsymbol{\omega_1}) ... \delta_{i,k}(\boldsymbol{\omega_k})$.

Définition 32 (Lexeur non déterministe)**.**
Le lexeur non déterministe associé au schéma de lexing $(\boldsymbol{\Sigma}, \boldsymbol{\Gamma}, (r_i)_{i\leq n}, (\delta_i)_{i\leq n})$ *est l'automate shift-action avec priorité dont :*
— *l'alphabet de token est* $\boldsymbol{\Sigma}$,
— *les états sont :*
  — *l'état initial* q_0,
  — *les couples* (i, q) *tels qui* $i \leq n$ *et* $q \in \mathcal{Q}_{A_{r_i}} - \{\bot\}$ *est un état de l'automate reconnaissant* r_i *(sauf* $\bot$ *que l'on enlève),*
— *les actions possibles* $\Delta := \{\delta'_i \mid i \leq n\}$, *ordonnés par* i, *sont sont les actions* δ'_i *qui sur le buffer* $\boldsymbol{u}$ *et le résiduel* $\boldsymbol{\omega}$ *vont renvoyer* $\delta_i(\boldsymbol{u})$ *sur un buffer de sortie, puis soit terminer si* $\boldsymbol{\omega} = \epsilon$, *soit relancer l'exécution du lexeur sur* $\boldsymbol{\omega}$,
— *l'état initial est* q_0,
— *les* ϵ*-transitions vont l'état initial vers chacun des états initiaux des automates des expressions régulières :*

$$\texttt{eps}(q_0) = \{(i, q_0) \mid i \leq n\}\,, \qquad \texttt{eps}((i,q)) = \emptyset\,,$$

— *les transitions sont celles des différents automates finis :*

$$\texttt{shift}(q_0, t) = \emptyset, \qquad \texttt{shift}((i,q),t) = \{(i, \texttt{trans}(q,t)) \mid \texttt{trans}(q,t) \neq \bot\},$$

— *les actions sont activées lorsque l'on arrive plus à lire un caractère :*

$$\texttt{action}(q_0, t) = \emptyset\,, \qquad \texttt{action}((i,q),t) = \{\delta_i \mid \texttt{trans}(q,t) = \bot\}\,.$$

Définition 33 (Lexeur)**.**
Le lexeur associé au schéma de lexing $(\mathbf{\Sigma}, \mathbf{\Gamma}, (r_i)_{i\leq n}, (\delta_i)_{i\leq n})$ *est la déterminisation et minimisation du lexeur non-déterministe.*

Theorem 2. *La fonction de lexing calculé par l'exécution du lexeur associé à un schéma est correcte pour le même schéma.*

6 Analyse Syntaxique

6.1 Parseur LR

6.1.1 Parseur pour une grammaire

Définition 34 (Parseur LR)**.**
Soit $\mathcal{G} = (\boldsymbol{\mathcal{T}}, \mathcal{N}, S, \tau)$ *une grammaire. Un parseur (resp. non-déterministe) LR pour* $\mathcal{G}$ *est un automate shift-action (resp. non-déterministe) dont :*

— *l'alphabet* $\boldsymbol{\mathcal{T}} \uplus \boldsymbol{\mathcal{N}}$ *est formé par les terminaux et non terminaux,*

— *les actions sont de deux types* $\Delta = \{\texttt{err}, \texttt{accept}\} \cup \{\texttt{reduce}_{N\mapsto\omega} \mid N \in \mathcal{N}\omega \in \tau(N)\}$ *:*

  — `err` *revoit toujours une erreur,* [5]

  — `accept` *renvoi le buffer* $\boldsymbol{u}$ *si* $u = S$ *et une erreur sinon,*

  — $\texttt{reduce}_{N\mapsto\omega}$ *revoit une erreur si* $u \neq v\omega$*, c.à.d. si le buffer* u *ne termine par par* ω*, si on a bien* $u = v\omega$ *alors on forme un arbre syntactique* $\boldsymbol{N} := N(\boldsymbol{\omega})$ *et on relance l'automate shift-action sur* $v\boldsymbol{N}\omega'$ *où* ω' *était le résiduel.*

— *l'action* `accept` *ne peut être renvoyé que en voyant la fin du mot :* $\texttt{accept} \in \texttt{action}(q, \boldsymbol{t}) \Rightarrow t = \$$.

Définition 35. *Un parseur (resp. non-déterministe) LR est correcte si :*

— *les accepts et reduces ne déclencheront jamais d'erreurs (on peut avoir des erreur* `err`*),*

— *un mot est accepté par le parseur si et seulement si il est accepté par sa grammaire,*

— *un mot accepté revoit un de ses arbres de dérivation dans la grammaire.*

6.1.2 Parseur canonique d'une grammaire

Définition 36 (Parseur (non déterministe) canonique)**.**
Soit $\mathcal{G} = (\mathcal{T}, \mathcal{N}, S, \tau)$ *une grammaire. Le parseur canonique de* $\mathcal{G}$ *est le parseur non-déterministe LR sur* $\mathcal{G}$ *donné par :*

5. on peut aussi mettre plusieurs erreurs plus expressives.

— *les états* $Q = \mathcal{N} \uplus \bigcup_{N \in \mathcal{L}} \tau_\bullet(N) \uplus \{(* \mapsto \bullet S), (* \mapsto S\bullet)\}$ *sont les non-terminaux, les règles partielles, et deux états particuliers se comportant comme des règles partielles spéciales : l'état initial* $q_0 = (* \mapsto \bullet S) \in L$*, l'état final* $\texttt{accept} = (* \mapsto S\bullet)$*,*

— *pas de shift ni d'action depuis un non-terminal :*

$$\texttt{shift}(N, \boldsymbol{t}) = \texttt{action}(N, \boldsymbol{t}) = \emptyset\,,$$

— *par contre il y a des* ϵ*-transitions :*

$$\texttt{epsilon}(N) = \{N \mapsto \bullet\omega \mid \omega \in \tau(N)\}\,,$$

— *pas d'action depuis une règle partielle non-final :*

$$\texttt{action}(N \mapsto \omega_1 \bullet \underline{c} \omega_2, \boldsymbol{t}) = \emptyset\,,$$

— *les transitions de shift depuis une règle partielle correspondent à avancer dans la règle :*

$$\texttt{shift}((N \mapsto \omega_1 \bullet \underline{c} \omega_2), \underline{\boldsymbol{c}'}) = \begin{cases} \{N \mapsto \omega_1 \underline{c} \bullet \omega_2\} & \text{si } \underline{c} = \underline{c}' \\ \emptyset & \text{sinon} \end{cases}\,,$$

— *l'ensemble des* ϵ*-transitions de depuis une règles partielle est le non-terminal qu'il faut lire si besoin :*

$$\texttt{epsilon}(N \mapsto \omega_1 \bullet \underline{c} \omega_2) = \begin{cases} \{\underline{c}\} & \text{si } \underline{c} \text{ est un non-terminal} \\ \emptyset & \text{si } \underline{c} \text{ est un terminal} \end{cases}\,,$$

— *pas de shift ni de epsilon depuis une règle partielle finale :*

$$\texttt{shift}(N \mapsto \omega\bullet, \boldsymbol{c}) = \texttt{epsion}(N \mapsto \omega\bullet) = \emptyset\,,$$

— *les reduce des règles partielle finales sont les réécriture indiquées par les états, la seul transition final est depuis la règle* $* \mapsto S\bullet$ *:*

$$\texttt{shift}((N \mapsto \omega\bullet), \underline{\boldsymbol{c}}) = \begin{cases} \{\texttt{reduce}_{N \mapsto \omega}\} & \text{si } N \neq * \\ \{\texttt{accept}\} & \text{si } N = * \text{ et } \underline{c} = \$ \\ \emptyset & \text{sinon} \end{cases}\,,$$

Définition 37 (Parseur spécial canonique)**.**
Il s'agit du parseur canonique d'une grammaire $\mathcal{G} = (\mathcal{T}, \mathcal{N}, S, \tau)$ *sauf que l'on restreint les reduces en utilisant les follows :*

$$\texttt{shift}((N \mapsto \omega\bullet), \underline{\boldsymbol{c}}) = \begin{cases} \{\texttt{reduce}_{N \mapsto \omega}\} & \text{si } N \neq * \text{ et } \underline{c} \in \texttt{follow}(N) \\ \{\texttt{accept}\} & \text{si } N = * \text{ et } \underline{c} = \$ \\ \emptyset & \text{sinon} \end{cases}\,,$$

Theorem 3. *Le parseur canonique de* $\mathcal{G}$ *est correcte (pour* $\mathcal{G}$*).*

6.1.3 (Psedo-)parseur LR_0

L'algorithme LR_0 est simple : il s'agit de psedo-déterminiser le parseur canonique de notre grammaire, on échoue si le résultat n'est pas déterministe.

Algorithme 11 (Construction du psedo-parseur LR_0)**.**
Donnée : une grammaire $\mathcal{G} = (\mathcal{T}, \mathcal{N}, S, \tau)$
Résultat : un parseur psedo-déterministe LR $\mathtt{LR}_0(\mathcal{G})$ sur $\mathcal{G}$,
Spécification : $\mathtt{LR}_0(\mathcal{G})$ est correcte pour $\mathcal{G}$.

Étape 1 : construire le parseur canonique de $\mathcal{G}$,
Étape 2 : le psedo-déterminiser,
Étape 3 : verifier qu'il n'y a pas de conflit,
Étape 4 (optionelle) : le minimiser.

Définition 38 (Parseur et grammaire LR_0)**.**
Une grammaire est LR_0 si $\mathtt{LR}_0(\mathcal{G})$, son psedo-parseur LR_0, se trouve être déterministe au sens de la Proposition 10.
Le parseur correspondant est nome le parseur LR_0 de la grammaire.

6.1.4 (Psedo-)parseur SLR

Pour l'algorithme SLR, on prend en compte les follows pour éviter quelques conflits :

Algorithme 12 (Construction du psedo-parseur SLR)**.**
Donnée : une grammaire $\mathcal{G} = (\mathcal{T}, \mathcal{N}, S, \tau)$
Résultat : un parseur psedo-déterministe LR $\mathtt{SLR}(\mathcal{G})$ sur $\mathcal{G}$,
Spécification : $\mathtt{SLR}(\mathcal{G})$ est correcte pour $\mathcal{G}$.

Étape 1 : construire le parseur spécial canonique de $\mathcal{G}$,
Étape 2 : le psedo-déterminiser,
Étape 3 : verifier qu'il n'y a pas de conflit,
Étape 4 (optionelle) : le minimiser.

Définition 39 (Parseur et grammaire SLR)**.**
Une grammaire est SLR si $\mathtt{SLR}(\mathcal{G})$, son psedo-parseur SLR, se trouve être déterministe au sens de la Proposition 10.
Le parseur correspondant est nommé le parseur SLR de la grammaire.

6.1.5 L'algorithme $\mathrm{LR}_1{}^{\dagger}$

Le principe est simple : on modifie la grammaire en dupliquant les non-terminaux de sorte à ce que chaque non-terminal n'ai qu'un seul follow.

Définition 40 (Grammaire LR_1-étendue)**.**
Soit $\mathcal{G} = (\mathcal{T}, \mathcal{N}, S, \tau)$ une grammaire.
La LR_1-extension de G est la grammaire $\mathtt{exten}_{\mathtt{LR}_1}(\mathcal{T}, \mathcal{N}_{\mathtt{LR}_1}, \tau_{\mathtt{LR}_1})$ définie par :

— *les non-terminaux sont dupliqués pour chaque follow :*

$$\mathcal{N}_{\mathtt{LR}_1} := \{(N,t) \in \mathcal{N} \times \mathcal{T} \mid t \in \mathtt{follow}(N)\}\,,$$

on écrira N_t *pour* (N,t)*,*

— *règles sont dupliquées et modifiées en conséquent :*

$$\tau_{\mathtt{LR}_1}(N_t) := \bigcup_{\omega \in \tau(N)} \omega_t$$

où $\omega_t \subseteq (\mathcal{T}_{\mathtt{LR}_1} \uplus \mathcal{N})_{\mathtt{LR}_1}$ *est définit par induction :*

$$(\omega s)_t := \{\omega' s \mid \omega' \in \omega_u\} \qquad (\omega M)_t := \left\{ \omega' M_t \;\middle|\; \omega' \in \bigcup_{t' \in first(Mt)} v_{t'} \right\}.$$

Proposition 12. *La* LR_1*-extension* $\mathtt{exten}_{\mathtt{LR}_1}(G)$ *d'une grammaire* G *à des singletons comme follow :*

$$\forall N_t \in \mathcal{N}_{\mathtt{LR}_1},\ \mathtt{follow}(N_t) = \{t\}\ .$$

Algorithme 13 (Construction du psedo-parseur LR_1)**.**
Donnée : une grammaire $\mathcal{G} = (\mathcal{T}, \mathcal{N}, S, \tau)$
Résultat : un parseur psedo-déterministe LR $\mathtt{LR}_1(\mathcal{G})$ *sur* $\mathcal{G}$*,*
Spécification : $\mathtt{LR}_1(\mathcal{G})$ *est correcte pour* $\mathcal{G}$*.*

Étape 1 : calculer les firsts et follows,
Étape 2 : calculer la LR_1*-extension* $\mathtt{exten}_{\mathtt{LR}_1}(\mathcal{G})$ *de* $\mathcal{G}$*,*
Étape 3 : construction du psedo-parseur SLR de $\mathtt{exten}_{\mathtt{LR}_1}(G)$*,*
Étape 4 : remplacer les reduces par des G-reduces en enlevant les annotations de follow,
Étape 5 : verifier qu'il n'y a pas de conflit.

Définition 41 (Parseur et grammaire LR_1)**.**
Une grammaire est LR_1 *si* $\mathtt{LR}_1(\mathcal{G})$*, son psedo-parseur* LR_1*, se trouve être déterministe au sens de la Proposition 10.*
Le parseur correspondant est nommé le parseur LR_1 *de la grammaire.*

6.1.6 L'algorithme LALR†

Cette fois, on modifie la grammaire en dupliquant les non-terminaux de sorte à ce qu'à chaque non-terminal N corresponde un seul shift dans le parseur résultant :

$$\forall N,\quad \exists! q,\quad \mathtt{shift}(q,N) \neq \emptyset$$

Remarquez que l'on a l'équivalence :

Proposition 13. *Dans une psedo-extention SLR, on a la propriété suivante :*

$$\mathtt{shift}(q,N) \neq \emptyset \quad \Leftrightarrow \quad \exists \omega, (N \mapsto \bullet\omega) \in q\ .$$

Une autre façon de décrire l'extension grammaticale est de dire qu'après la déterminisation du parseur LR spécial canonique, chaque non-terminal N ne se retrouve que dans un seul état du psedo-parseur SLR.

Définition 42 (Grammaire LALR-étendue)**.**
Soit $\mathcal{G} = (\mathcal{T}, \mathcal{N}, S, \tau)$ une grammaire et $\texttt{SLR}(\mathcal{G}) = (\mathcal{T}, \mathcal{N}, q_0, \bot, \texttt{shift}, \texttt{action})$ le psedo-parseur SLR de G.
La LALR-extension de G est la grammaire $\texttt{exten}_{\texttt{LALR}}(\mathcal{T}, \mathcal{N}_{\texttt{LALR}}, \tau_{\texttt{LALR}})$ définie par :

les non-terminaux sont dupliqués pour chaque utilisation dans $\mathcal{A}$:

$$\mathcal{N}_{\texttt{LALR}} := \{(N, q) \in \mathcal{N} \times \mathcal{T} \mid \texttt{shift}(q, N) \neq \emptyset\},$$

on écrira N_q pour (N, q),

règles sont dupliquées et modifiées en conséquent :

$$\tau_{\texttt{LR}_1}(N_q) := \bigcup_{\omega \in \tau(N)} \texttt{LA}(q, N \mapsto \bullet\omega)$$

où l'ensemble $\texttt{LA}(q, N \mapsto \omega'\bullet\omega)$ est définit par induction pour tout $(N \mapsto \omega'\bullet\omega) \in q$:

- $\texttt{LA}(q, N \mapsto \omega'\bullet) = \epsilon$,
- $\texttt{LA}(q, N \mapsto \omega' \bullet t\omega) = t.\texttt{LA}(q', N \mapsto \omega' t \bullet \omega)$ *avec* $\{q'\} = \texttt{shift}(q, t)$ *si t est un terminal,*
- $\texttt{LA}(q, N \mapsto \omega' \bullet M\omega) = M_q.\texttt{LA}(q', N \mapsto \omega' M \bullet \omega)$ *avec* $\{q'\} = \texttt{shift}(q, M)$ *si M est un non-terminal.*

Proposition 14. $\texttt{SLR}(\texttt{exten}_{\texttt{LALR}}(\mathcal{G}))$ *est au pire aussi gros (en nombre d'états) que* $\texttt{SLR}(\mathcal{G})$.

Algorithme 14 (Construction du psedo-parseur LR_1)**.**
Donnée : une grammaire $\mathcal{G} = (\mathcal{T}, \mathcal{N}, S, \tau)$
Résultat : un parseur psedo-déterministe LR $\texttt{LALR}(\mathcal{G})$ sur $\mathcal{G}$,
Spécification : $\texttt{LALR}(\mathcal{G})$ est correcte pour $\mathcal{G}$.

Étape 1 : calculer $\texttt{SLR}(\mathcal{G})$,le psedo-parseur SLR de $\mathcal{G}$,
Étape 2 : calculer $\texttt{exten}_{\texttt{LALR}}(\mathcal{G})$ la LALR-extension de $\mathcal{G}$,
Étape 3 : construction du psedo-parseur SLR de $\texttt{exten}_{\texttt{LR}_1}(G)$,
Étape 4 : remplacer les reduces par des G-reduces en enlevant les annotations,
Étape 5 : verifier qu'il n'y a pas de conflit.

Définition 43 (Parseur et grammaire LALR)**.**
Une grammaire est LALR si $\texttt{LALR}(\mathcal{G})$, son psedo-parseur LALR, se trouve être déterministe au sens de la Proposition 10.
Le parseur correspondant est nommé le parseur LR_1 de la grammaire.