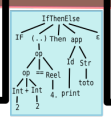
Rappel : Analyse Syntaxique / Parsing

Stream de Tokens

```
IF,ParG,
Int(2),Plus,
Int(2),Eg,
Reel(4.0),
ParD,THEN,
Id(print),
Str(toto)
```

Analyse Syntaxique

Construction de l'arbre syntaxique



Simplification / abstraction

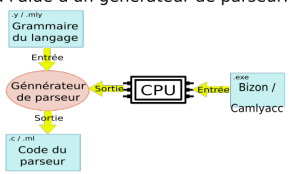
Deux types de parseurs

LL:
"devine" le noeud
que l'on va lire

LR :
"devine" le noeud
que l'on a lu

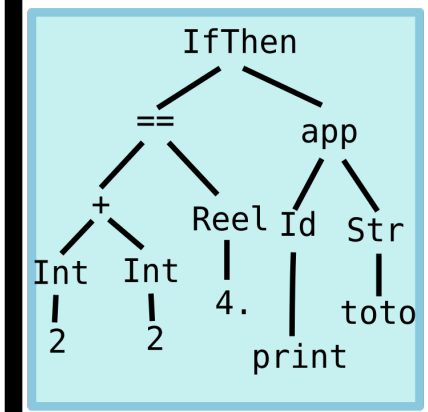
Comment faire un parseur ?

- à la main...
- à l'aide d'un générateur de parseur:

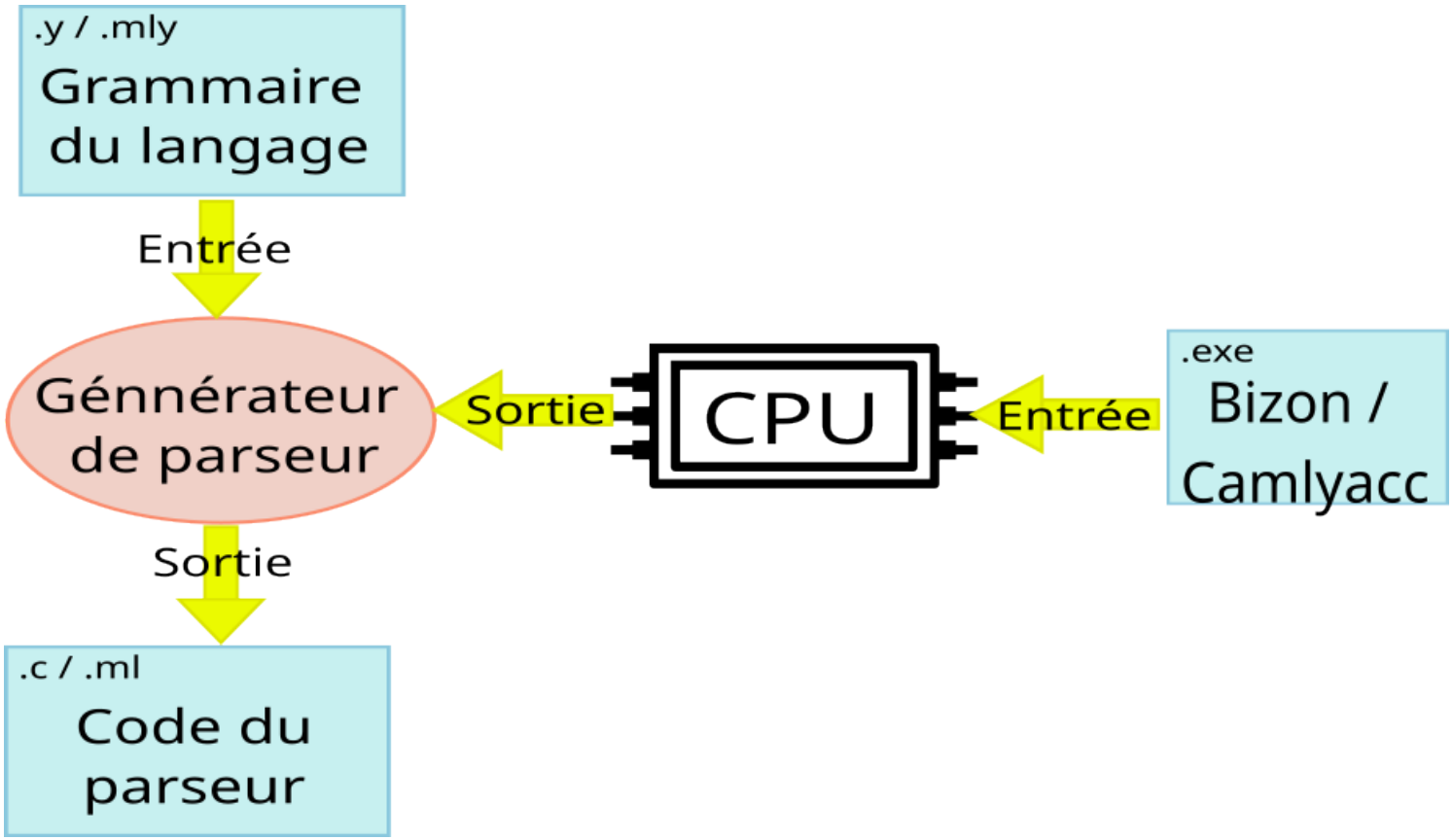


Parseur

Arbre syntaxique abstrait (AST)



Rappel : Générateur de parseur



Un exemple de grammaire (projet)

⟨programme⟩ ::= ⟨commande⟩ | ⟨commande⟩ ⟨programme⟩

⟨commande⟩ ::= `if (` ⟨expression⟩ `)` ⟨commande⟩ `else` ⟨commande⟩
| ⟨expression⟩ `;`
| `while (` ⟨expression⟩ `)` ⟨commande⟩
| `for(`⟨expression⟩`;`⟨expression⟩`;`⟨expression⟩`)`⟨commande⟩
| `;`
| `{`⟨programme⟩`}`

⟨expression⟩ ::= ⟨NUMBER⟩ | ⟨BOOLEEN⟩ | ⟨IDENT⟩
| `(`⟨expression⟩`)`
| ⟨expression⟩ `+` ⟨expression⟩
| ⟨expression⟩ `-` ⟨expression⟩
| ⟨expression⟩ `*` ⟨expression⟩
| ⟨expression⟩ `/` ⟨expression⟩
| ⟨expression⟩ `==` ⟨expression⟩
| `-` ⟨expression⟩
| ⟨IDENT⟩ `=` ⟨expression⟩

Terminaux et Non-terminaux + méta-syntax

⟨programme⟩ ::= ⟨commande⟩ | ⟨commande⟩ ⟨programme⟩

⟨commande⟩ ::= `if (` ⟨expression⟩ `)` ⟨commande⟩ `else` ⟨commande⟩
| ⟨expression⟩`;`
| `while (` ⟨expression⟩ `)` ⟨commande⟩
| `for(`⟨expression⟩`;`⟨expression⟩`;`⟨expression⟩`)`⟨commande⟩
| `;`
| `{`⟨programme⟩`}`

⟨expression⟩ ::= ⟨NUMBER⟩ | ⟨BOOLEEN⟩ | ⟨IDENT⟩
| `(`⟨expression⟩`)`
| ⟨expression⟩ `+` ⟨expression⟩
| ⟨expression⟩ `-` ⟨expression⟩
| ⟨expression⟩ `*` ⟨expression⟩
| ⟨expression⟩ `/` ⟨expression⟩
| ⟨expression⟩ `==` ⟨expression⟩
| `-` ⟨expression⟩
| ⟨IDENT⟩ `=` ⟨expression⟩

Terminaux et Non-terminaux

Les Terminaux

Ce sont les (noms des) tokens issus du lexeur !

Exemples : NOMBRE, IDENT, IF, STRING...

pour plus de lisibilité on utilise souvent les notations d'origines pour les singletons (mots-clefs, symboles), et des majuscules entre brackets ⟨VAR⟩ pour les autres.

Les Non-Terminaux

Ce sont les grandes classes structurant une phrase

Exemples : expressions, commandes, groupe nominal, COI...

pour plus de lisibilité on utilise des noms en minuscule entre brackets ⟨expression⟩

Forme alt. : séquence de règles

⟨programme⟩ ↦ ⟨commande⟩
⟨programme⟩ ↦ ⟨commande⟩ ⟨programme⟩
⟨commande⟩ ↦ `if (` ⟨expression⟩ `)` ⟨commande⟩ `else` ⟨commande⟩
⟨commande⟩ ↦ ⟨expression⟩`;`
⟨commande⟩ ↦ `while (` ⟨expression⟩ `)` ⟨commande⟩
⟨commande⟩ ↦ `for(`⟨expression⟩`;`⟨expression⟩`;`⟨expression⟩`)`⟨commande⟩
⟨commande⟩ ↦ `;`
⟨commande⟩ ↦ `{`⟨programme⟩`}`
⟨expression⟩ ↦ ⟨NUMBER⟩
⟨expression⟩ ↦ ⟨BOOLEEN⟩
⟨expression⟩ ↦ ⟨IDENT⟩
⟨expression⟩ ↦ `(`⟨expression⟩`)`
⟨expression⟩ ↦ ⟨expression⟩ `+` ⟨expression⟩
⟨expression⟩ ↦ ⟨expression⟩ `-` ⟨expression⟩
⟨expression⟩ ↦ ⟨expression⟩ `*` ⟨expression⟩
⟨expression⟩ ↦ ⟨expression⟩ `/` ⟨expression⟩
⟨expression⟩ ↦ ⟨expression⟩ `==` ⟨expression⟩
⟨expression⟩ ↦ `-` ⟨expression⟩
⟨expression⟩ ↦ ⟨IDENT⟩ `=` ⟨expression⟩

Forme alt. : séquence de règles

⟨programme⟩ ↦ ⟨commande⟩
⟨programme⟩ ↦ ⟨commande⟩ ⟨programme⟩
⟨commande⟩ ↦ `if (` ⟨expression⟩ `)` ⟨commande⟩ `else` ⟨commande⟩
⟨commande⟩ ↦ ⟨expression⟩`;`
⟨commande⟩ ↦ `while (` ⟨expression⟩ `)` ⟨commande⟩
⟨commande⟩ ↦ `for(`⟨expression⟩`;`⟨expression⟩`;`⟨expression⟩`)`⟨commande⟩
⟨commande⟩ ↦ `;`
⟨commande⟩ ↦ `{`⟨programme⟩`}`
⟨expression⟩ ↦
⟨expression⟩
⟨expression⟩
⟨expression⟩
⟨expression⟩ ↦
⟨expression⟩ ↦ ⟨expression⟩ `-` ⟨expression⟩
⟨expression⟩ ↦ ⟨expression⟩ `*` ⟨expression⟩
⟨expression⟩ ↦ ⟨expression⟩ `/` ⟨expression⟩
⟨expression⟩ ↦ ⟨expression⟩ `==` ⟨expression⟩
⟨expression⟩ ↦ `-` ⟨expression⟩
⟨expression⟩ ↦ ⟨IDENT⟩ `=` ⟨expression⟩

Associe un non-terminal
et un mot composé de
terminaux et non-terminaux

Définition formelle

Une grammaire est donnée par

- Un ensemble $\mathcal{T}$ de terminaux,
- Un ensemble $\mathcal{N}$ de non-terminaux,
- Un non-terminal principal $S \in \mathcal{N}$,
- Une relation $(\mapsto) \subseteq \mathcal{N} \times (\mathcal{T} \uplus \mathcal{N})^*$ décrivant les règles.

Attention au mot vide

Une règle $\langle \text{arguments} \rangle \mapsto \epsilon$
associe le non terminal $\langle \text{arguments} \rangle$ au mot vide (et non au caractère ϵ !)

Les règles ne regardent que le nom des non-terminaux

Les noms terminaux $\boldsymbol{t} \in \mathcal{T}$ sont des tokens, mais les dans les règles on ne regarde que leur noms $t \in \mathcal{T}$.

Version plus compacte

On associe une regexp aux non terminaux

À chaque non terminal $N \in \mathcal{N}$ on associe une expression régulière :

$$\tau : \mathcal{N} \to \texttt{RegExp}(\mathcal{T} \uplus \mathcal{N})$$

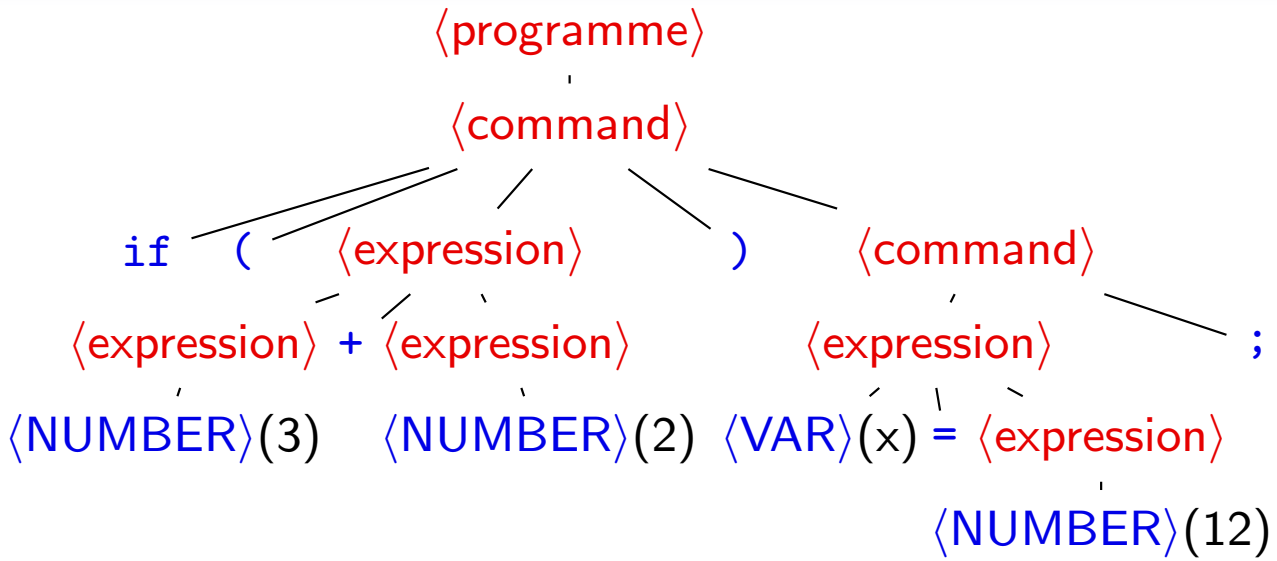
En faisant ça, on peut écrire des règles plus compactes :

$$\begin{array}{lll} \langle \text{commande} \rangle & ::= & \mid \texttt{function}\ \langle \text{IDENT} \rangle\ (\langle \text{arguments} \rangle)\ \{\ \langle \text{command} \rangle^*\ \} \\ & & \mid \texttt{Class}\ \langle \text{IDENT} \rangle\ (\texttt{extends}\ \langle \text{expression} \rangle)^?\ \langle \texttt{methode} \rangle^* \end{array}$$

Utilisé par JavaCC ou Antler par exemple

Avantage : Plus facile à écrire.
Défaut : Priorités/associativités plus difficiles à définir.

Arbre Syntaxique (def informelle)



Un arbre syntaxique est un arbre :

- Dont les noeuds internes sont des non-terminaux
- Dont les feuilles sont des terminaux
- Dont les fils d'un non-terminal N correspondent à une règle sur N.

Attention, avec cette définition, un noeud interne peut ne pas avoir de fils (règle vers le mot vide) et ne pas être une feuille.

Arbre Syntaxique (def formelle)

Soit $\mathcal{G} = (\mathcal{T}, \mathcal{N}, S, \tau)$ une grammaire.

On définit les arbres syntactiques de $\mathcal{G}$ comme

Un ensemble $\mathcal{N}$ de tokens/noeuds de noms $\mathcal{N}$ définit par point fixe :

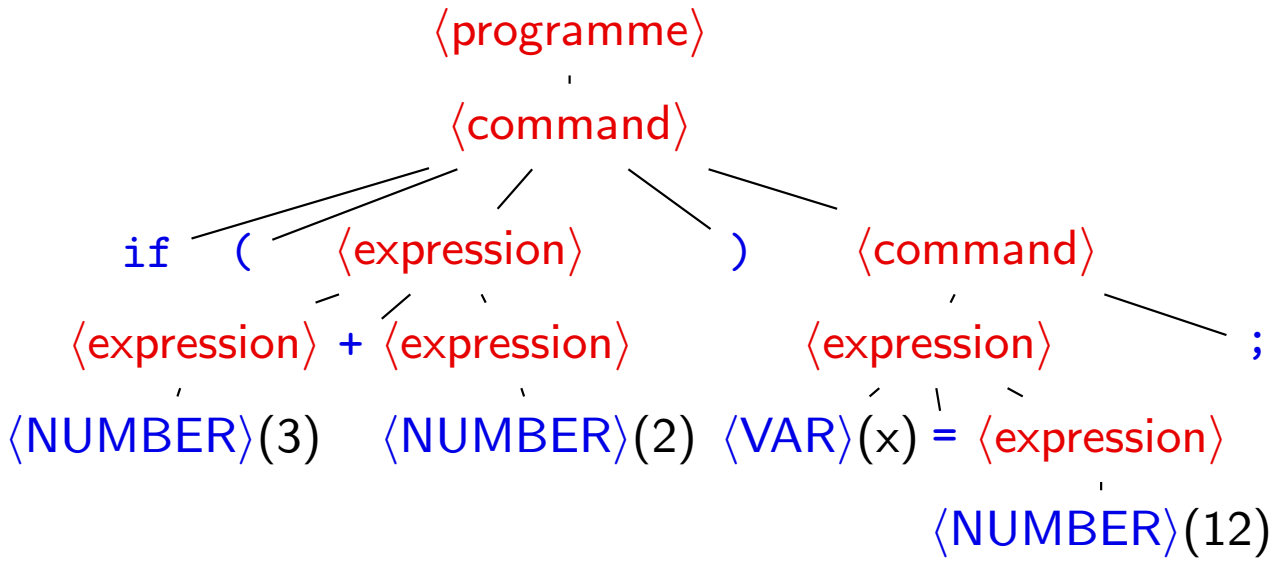
- Ensemble : $\mathcal{N} \subseteq \mathcal{N} \times (\mathcal{T} \uplus \mathcal{N})^*$,
- Initialisation : $\mathcal{N} = \emptyset$,
- Pas :

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

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

Mot reconnu par un arbre syntactique

C'est le mot lu sur les feuilles de gauche à droite.

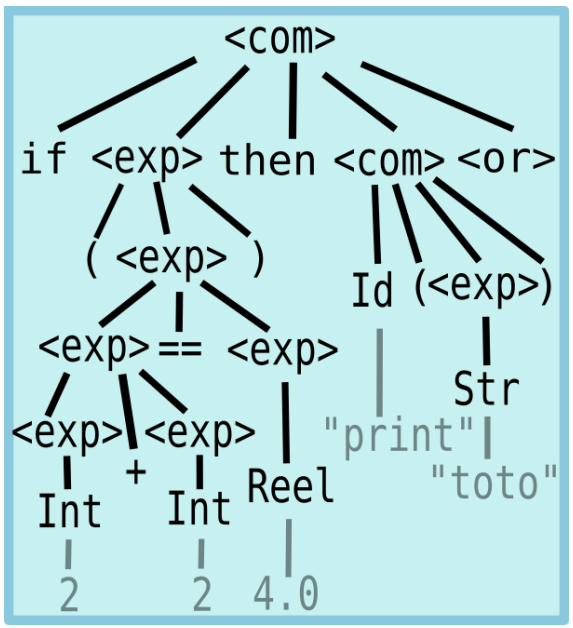


Reconnait :
⟨if⟩ ⟨(⟩ ⟨NUMBER⟩(3) ⟨+⟩ ⟨NUMBER⟩(2) ⟨)⟩ ⟨VAR⟩(x) ⟨=⟩
⟨NUMBER⟩(12) ⟨;⟩

Arbre syntaxique abstrait (AST)

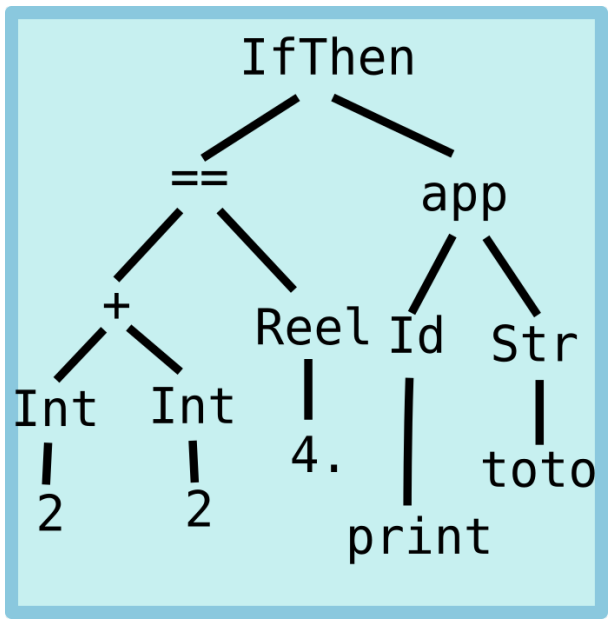
C'est une simplification arbitraire de l'arbre syntaxique qui enlève le superflu :

- suppression des parenthèses,
- remplacer les nom de non-terminaux par le nom de la règle utilisée
- suppression des tokens sans contenus.



Arbre Syntaxique

Devient



AST

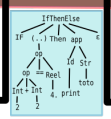
Rappel : Analyse Syntaxique / Parsing

Stream de Tokens

```
IF,ParG,
Int(2),Plus,
Int(2),Eg,
Reel(4.0),
ParD,THEN,
Id(print),
Str(toto)
```

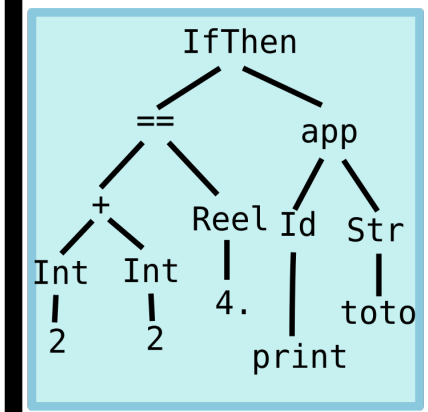
Analyse Syntaxique

Construction de l'arbre syntaxique



Simplification / abstraction

Arbre syntaxique abstrait (AST)



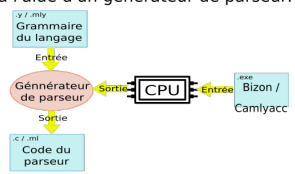
Deux types de parseurs

LL:
"devine" le noeud
que l'on va lire

LR :
"devine" le noeud
que l'on a lu

Comment faire un parseur ?

- à la main...
- à l'aide d'un générateur de parseur:



Parseur

De la théorie (TD) à la pratique (TP)

TD : pas d'AST

En TD, on se concentre sur la partie complexe; le décodage du ST.
On considère donc que l'arbre généré est le ST pour simplifier l'algo.

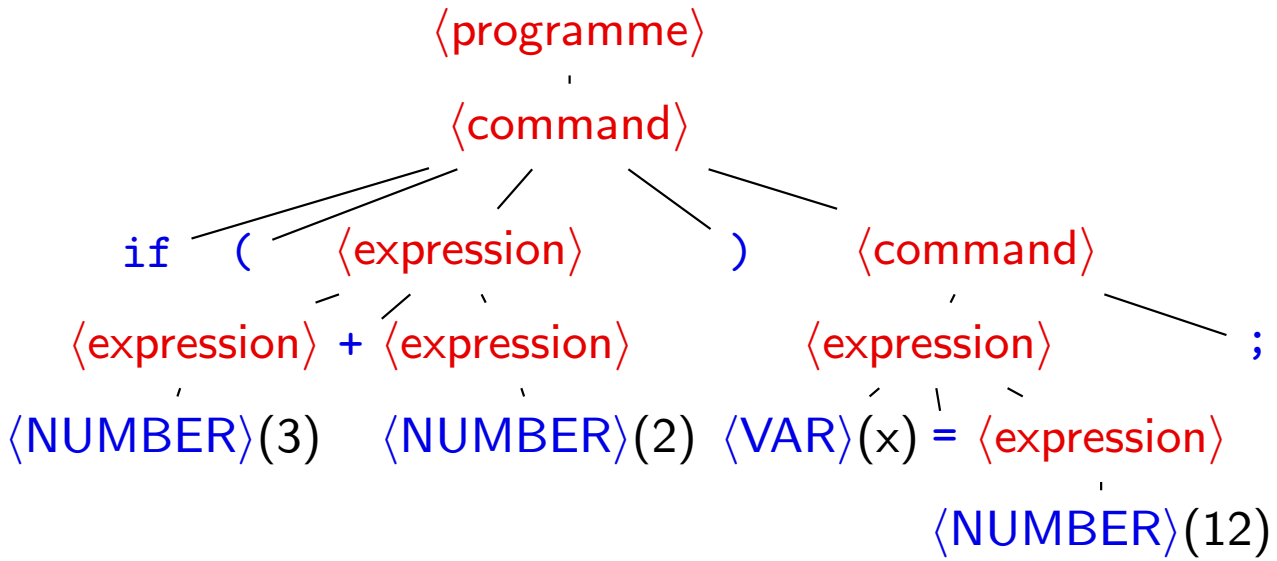
TP : Abstraction à la volée

En pratique, le parseur décode bien l'arbre syntaxique mais crée directement l'AST. (Les actions de l'automate sont plus complexe.)

Nous nous concentrons ici sur la théorie.

Mot reconnu par un arbre syntactique

C'est le mot lu sur les feuilles de gauche à droite.



Reconnait :
⟨if⟩ ⟨(⟩ ⟨NUMBER⟩(3) ⟨+⟩ ⟨NUMBER⟩(2) ⟨)⟩ ⟨VAR⟩(x) ⟨=⟩
⟨NUMBER⟩(12) ⟨;⟩

Le but d'un parseur

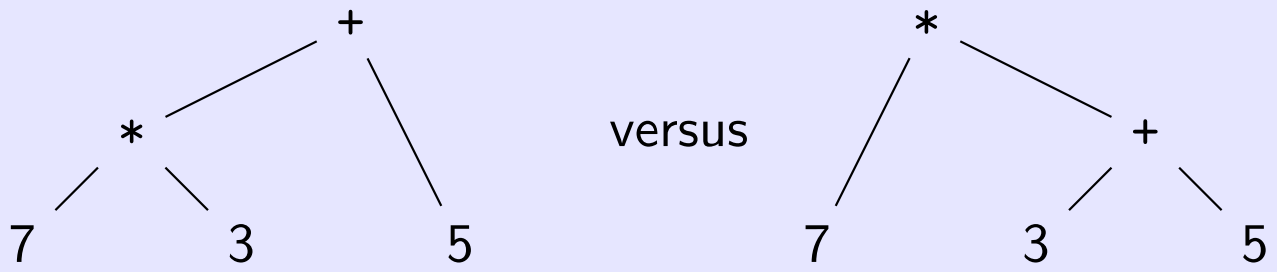
Trouver l'arbre qui reconnaît le mot en entrée

Problème : Ambigüité

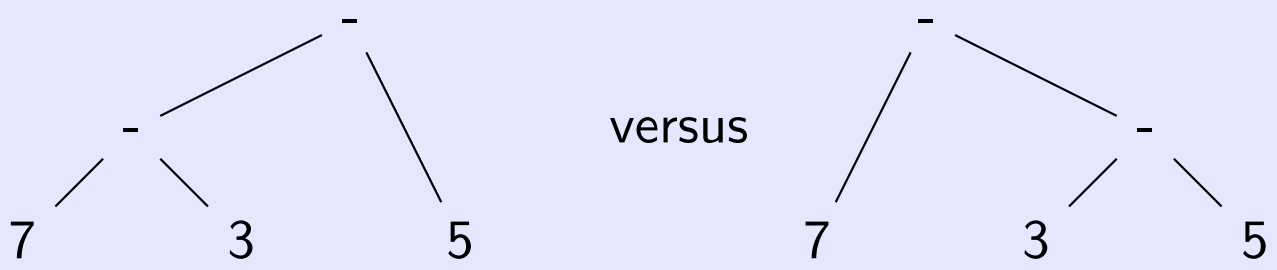
Si plusieurs arbres le reconnaissent, que fait-on ?

Les cas d'ambigüité classiques (AST)

Priorité



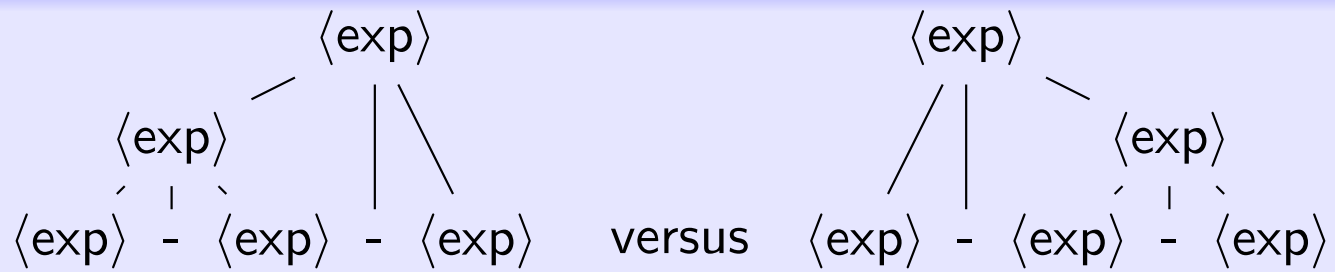
Associativité



Les cas d'ambigüité classiques (ST)

Priorité

Associativité



Quelques remarques

Même si l'opération est associative, le compilo ne le sait pas

Il faut bien construire un arbre...
Associativité gauche par défaut pour les générateurs LL (JavaCC).

Attention, dans le projet, le + et le $*$ ne sont pas associatifs.

On peut faire pareil avec des opérateurs d'autres arités et des groupes d'opérateurs

Quelques remarques

Même si l'opération est associative, le compilo ne le sait pas

Il faut bien construire un arbre...
Associativité gauche par défaut pour les générateurs LL (JavaCC).

Attention, dans le projet, le + et le $*$ ne sont pas associatifs.

On peut faire pareil avec des opérateurs d'autres arités et des groupes d'opérateurs

`!b&&c` $\rightsquigarrow$! prioritaire sur &&

Quelques remarques

Même si l'opération est associative, le compilo ne le sait pas

Il faut bien construire un arbre...
Associativité gauche par défaut pour les générateurs LL (JavaCC).

Attention, dans le projet, le + et le $*$ ne sont pas associatifs.

On peut faire pareil avec des opérateurs d'autres arités et des groupes d'opérateurs

`!b&&c`	$\rightsquigarrow$	! prioritaire sur &&
`if b then if c then ; else ;`	$\rightsquigarrow$	ifthen prioritaire sur ifthenelse

Quelques remarques

Même si l'opération est associative, le compilo ne le sait pas

Il faut bien construire un arbre...
Associativité gauche par défaut pour les générateurs LL (JavaCC).

Attention, dans le projet, le + et le $*$ ne sont pas associatifs.

On peut faire pareil avec des opérateurs d'autres arités et des groupes d'opérateurs

`!b&&c`	$\rightsquigarrow$	! prioritaire sur &&
`if b then if c then ; else ;`	$\rightsquigarrow$	ifthen prioritaire sur ifthenelse
`a?b:c?e:f`	$\rightsquigarrow$	`_?_:_` associatif à gauche

Quelques remarques

Même si l'opération est associative, le compilo ne le sait pas

Il faut bien construire un arbre...
Associativité gauche par défaut pour les générateurs LL (JavaCC).

Attention, dans le projet, le + et le $*$ ne sont pas associatifs.

On peut faire pareil avec des opérateurs d'autres arités et des groupes d'opérateurs

`!b&&c`	$\rightsquigarrow$	! prioritaire sur &&
`if b then if c then ; else ;`	$\rightsquigarrow$	ifthen prioritaire sur ifthenelse
`a?b:c?e:f`	$\rightsquigarrow$	_?_:_ associatif à gauche
`a+b-c+d`	$\rightsquigarrow$	+,- asso. à gauche ensembles

Quelques remarques

Même si l'opération est associative, le compilo ne le sait pas

Il faut bien construire un arbre...
Associativité gauche par défaut pour les générateurs LL (JavaCC).

Attention, dans le projet, le + et le $*$ ne sont pas associatifs.

On peut faire pareil avec des opérateurs d'autres arités et des groupes d'opérateurs

`!b&&c`	$\rightsquigarrow$	! prioritaire sur &&
`if b then if c then ; else ;`	$\rightsquigarrow$	ifthen prioritaire sur ifthenelse
`a?b:c?e:f`	$\rightsquigarrow$	_?_:_ associatif à gauche
`a+b-c+d`	$\rightsquigarrow$	+,- asso. à gauche ensembles
`a>b>c` (en python)	$\rightsquigarrow$	_>_>_ prioritaire sur _>_

Quelques remarques

Même si l'opération est associative, le compilo ne le sait pas

Il faut bien construire un arbre...
Associativité gauche par défaut pour les générateurs LL (JavaCC).

Attention, dans le projet, le + et le $*$ ne sont pas associatifs.

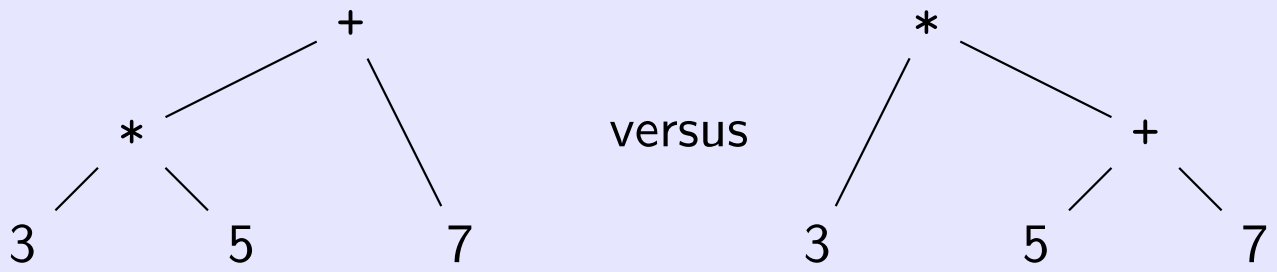
On peut faire pareil avec des opérateurs d'autres arités et des groupes d'opérateurs

`!b&&c`	$\rightsquigarrow$	! prioritaire sur &&
`if b then if c then ; else ;`	$\rightsquigarrow$	ifthen prioritaire sur ifthenelse
`a?b:c?e:f`	$\rightsquigarrow$	_?_:_ associatif à gauche
`a+b-c+d`	$\rightsquigarrow$	+,- asso. à gauche ensembles
`a>b>c` (en python)	$\rightsquigarrow$	_>_>_ prioritaire sur _>_

Le sense exacte de la priorité et assiciativité sera expliqué lors de l'élimination des conflits du parseur.

Régler les problèmes de priorité

Priorité



∗ prioritaire sur +

\+ prioritaire sur ∗

Solution 1

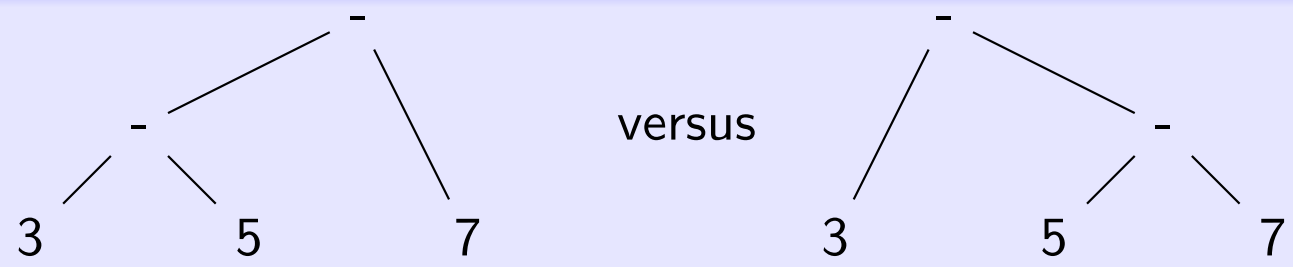
Changer la grammaire, avec un nouveau non-terminal par niveau de priorité (voire exo 1 du TP).

Solution 2

Utiliser un générateur de parseur dans lequel on peut indiquer la priorité (voir exo 3 du TP OCaml ou C).

Régler les problèmes d'associativité

Associativité



(-) associatif à gauche

(-) associatif à droite

Solution 1

Changer la grammaire, avec un non-terminal intermédiaire (cf TP1 exo1)

Solution 2

Utiliser un générateur de parseur dans lequel on peut indiquer l'associativité gauche ou droite (voir exo 3 du TP1 OCaml ou C).

Solution 3

Utiliser un générateur de parseur avec grammaire compacte (TP1 java)

Exemple de parseur (OCamlyacc)

```
%token <int> NOMBRE
%token PLUS MOINS FOIS GPAREN DPAREN EOL
%left PLUS MOINS
%left FOIS DIV
%nonassoc UMOINS
%type <AST.expression_a> main expression
%start main
%%
main:
   expression EOL                    { $1 }              ;
expression:
   expression PLUS expression        { Plus ($1,$3) }
 | expression MOINS expression       { Moins($1,$3) }
 | expression FOIS expression        { Mult ($1,$3) }
 | GPAREN expression DPAREN          { $2 }
 | MOINS expression %prec UMOINS     { Neg $2 }
 | NOMBRE                            { Num $1 }          ;
```

Exem (cc)

Tokens/terminaux contenant un entier

Tokens/terminaux sans contenus

associativité à gauche et priorité (ordre)

non terminaux

règles

AST: création noeud “+”

AST: oubli parenthèse

```
%token <int> NOMBRE
%token PLUS MOINS FOIS GPAREN DPAREN EOL
%left PLUS MOINS
%left FOIS DIV
%nonassoc UMOINS
%type <AST.expression_a> main expression
%start main
%%
main:
    expression EOL                  { ;
expression:
    expression PLUS expression      { Plus (($1,$3) }
  | expression MOINS expression     { Moins($1,$3) }
  | expression FOIS expression      { Mult (($1,$3) }
  | GPAREN expression DPAREN        { $2 }
  | MOINS expression %prec UMOINS   { Neg $2
  | NOMBRE                          { Num $1 }          ;
```

Exemple de parseur (Bison)

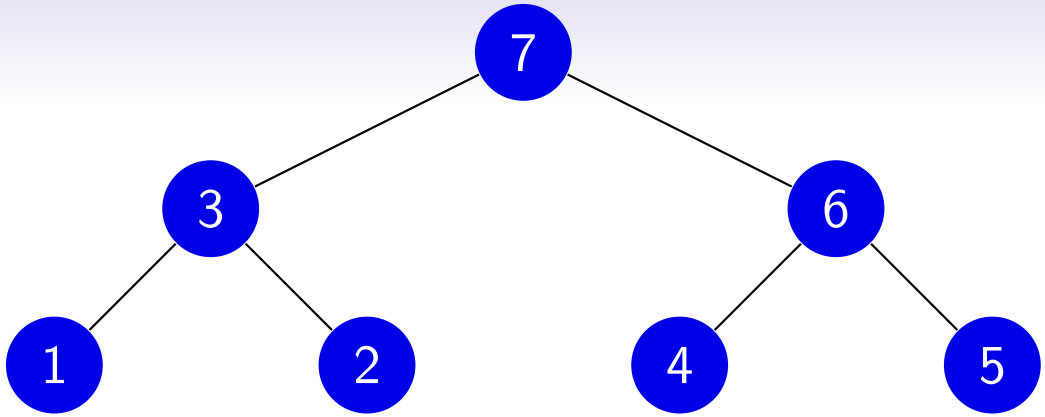
```
%parse-param {struct ExpressionA* ast}
%union { struct ExpressionA* expA; int num; }
%type <expA> expression
%token <num> NOMBRE
%left '+' '-'
%left '*' '/'
%nonassoc MU
%%
resultat:   expression          {*ast = *$1;}
expression:
  expression '+' expression {$$=newExpression('+',$1,$3,0);}
| expression '-' expression {$$=newExpression('-',$1,$3,0);}
| expression '*' expression {$$=newExpression('*',$1,$3,0);}
| expression '/' expression {$$=newExpression('/',$1,$3,0);}
| '(' expression ')'        {$$=$2;}
| '-' expression %prec MU   {$$=newExpression('-',$2,NULL,0);}
| NOMBRE                    {$$=newExpression('0',NULL,NULL,$1);
%%
```

Exemple de parseur (JavaCC)

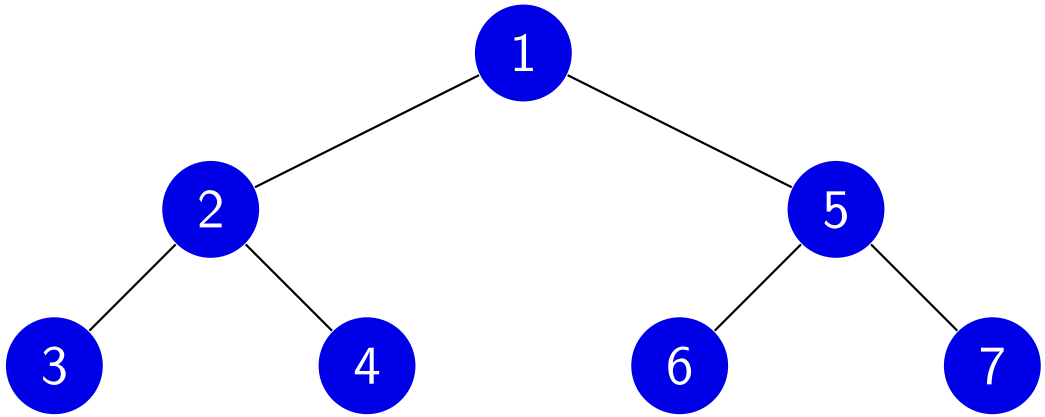
```
SKIP : { " " | "\t" }
TOKEN : { < NOMBRE: ["1"-"9"] (["0"-"9"])* > | < EOL: "\n" > }

AST.ExpressionA mainNT () :
{ AST.ExpressionA e; }
{
    e=expression() <EOL>          {return e;}
}
AST.ExpressionA expression () :
{ AST.ExpressionA exp;
  AST.ExpressionA e;
}
{
    e=terme()                     {exp = e;}
    ( "+" (e=terme())             {exp = new AST.Plus(exp,e);}
    | "-" (e=terme())             {exp = new AST.Moins(exp,e);}
    )*                            {return exp;}
}
```

Suite JavaCC

```
AST.ExpressionA terme () :
{ AST.ExpressionA exp;
  AST.ExpressionA e;
}
{
    e=facteur()                    {exp = e;}
    ( "*" (e=facteur())            {exp = new AST.Mult(exp,e);}
    | "/" (e=facteur())            {exp = new AST.Div(exp,e);}
    )*                             {return exp;}
}
AST.ExpressionA facteur () :
{ AST.ExpressionA e;
  Token t;
}
{
    "(" (e=expression()) ")"       {return e;}
  | "-" (e=facteur())              {return new AST.Neg(e);}
  | t=<NOMBRE>                     {return new AST.Num(Integer.parse
}
```



lecture	décodage	encodage	modification
profondeur gauche remontant	Left-right (LR)	suffix	leftmost-innermost



lecture	décodage	encodage	modification
profondeur gauche descendant	Left-Left (LL)	prefix	leftmost-outermost

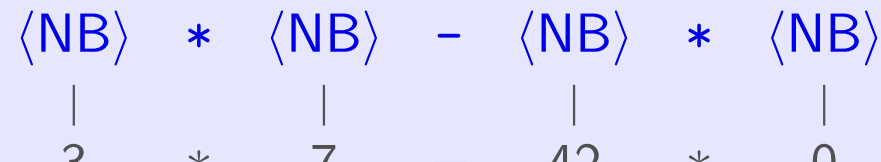
Parseur LR en action

Construction de l'abre profondeur gauche remontant

Grammaire :

$$
\begin{array}{rcl}
\langle \text{expr} \rangle & ::= & \langle \text{expr} \rangle \ \texttt{-} \ \langle \text{expr} \rangle \\
& | & \langle \text{expr} \rangle \ \texttt{*} \ \langle \text{expr} \rangle \\
& | & \texttt{(} \ \langle \text{expr} \rangle \ \texttt{)} \\
& | & \texttt{-} \ \langle \text{expr} \rangle \\
& | & \langle \text{NB} \rangle
\end{array}
$$

Reconstruction de l'arbre :



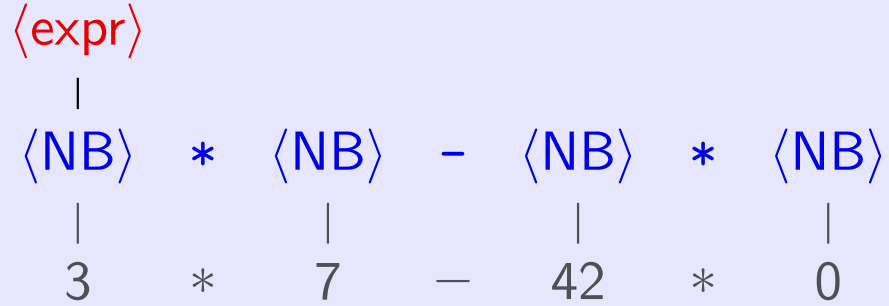
Parseur LR en action

Construction de l'abre profondeur gauche remontant

Grammaire :

$$
\begin{array}{rcl}
\langle \text{expr} \rangle & ::= & \langle \text{expr} \rangle - \langle \text{expr} \rangle \\
& | & \langle \text{expr} \rangle * \langle \text{expr} \rangle \\
& | & (\langle \text{expr} \rangle) \\
& | & - \langle \text{expr} \rangle \\
& | & \langle \text{NB} \rangle
\end{array}
$$

Reconstruction de l'arbre :



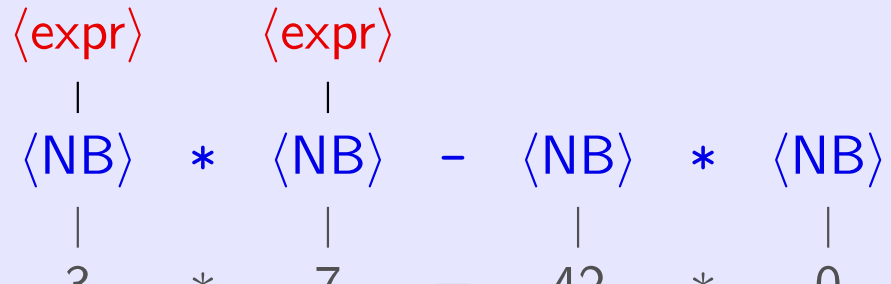
Parseur LR en action

Construction de l'abre profondeur gauche remontant

Grammaire :

$$
\begin{array}{rcl}
\langle \text{expr} \rangle & ::= & \langle \text{expr} \rangle - \langle \text{expr} \rangle \\
& | & \langle \text{expr} \rangle * \langle \text{expr} \rangle \\
& | & (\langle \text{expr} \rangle) \\
& | & - \langle \text{expr} \rangle \\
& | & \langle \text{NB} \rangle
\end{array}
$$

Reconstruction de l'arbre :



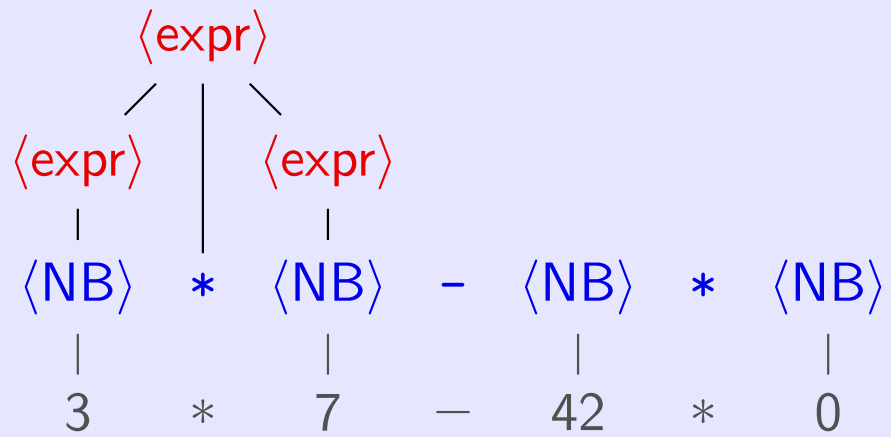
Parseur LR en action

Construction de l'abre profondeur gauche remontant

Grammaire :

$$
\begin{aligned}
\langle \text{expr} \rangle \ ::= \quad & \langle \text{expr} \rangle \ \texttt{-} \ \langle \text{expr} \rangle \\
\mid \ & \langle \text{expr} \rangle \ \texttt{*} \ \langle \text{expr} \rangle \\
\mid \ & \texttt{(} \ \langle \text{expr} \rangle \ \texttt{)} \\
\mid \ & \texttt{-} \ \langle \text{expr} \rangle \\
\mid \ & \langle \text{NB} \rangle
\end{aligned}
$$

Reconstruction de l'arbre :



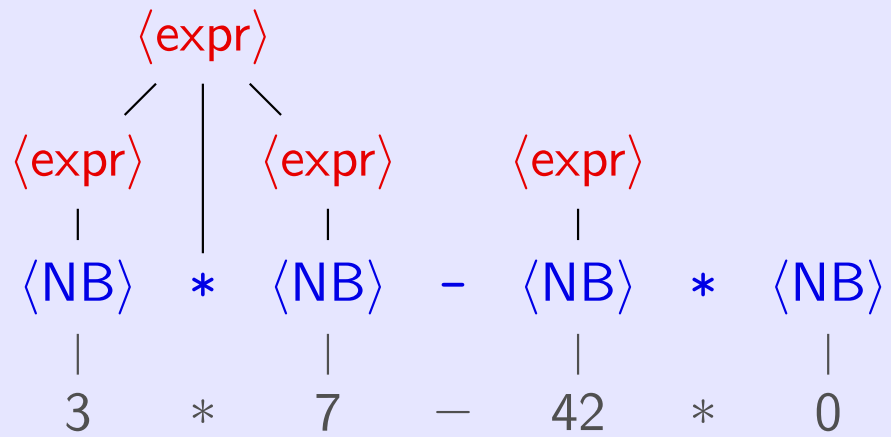
Parseur LR en action

Construction de l'abre profondeur gauche remontant

Grammaire :

$$
\begin{aligned}
\langle \text{expr} \rangle \ ::= \ & \langle \text{expr} \rangle \ \texttt{-} \ \langle \text{expr} \rangle \\
| \ & \langle \text{expr} \rangle \ \texttt{*} \ \langle \text{expr} \rangle \\
| \ & \texttt{(} \ \langle \text{expr} \rangle \ \texttt{)} \\
| \ & \texttt{-} \ \langle \text{expr} \rangle \\
| \ & \langle \text{NB} \rangle
\end{aligned}
$$

Reconstruction de l'arbre :



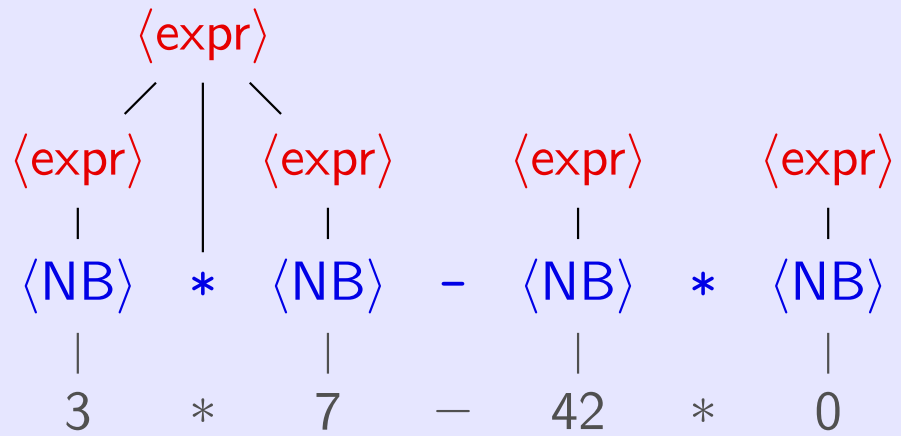
Parseur LR en action

Construction de l'abre profondeur gauche remontant

Grammaire :

$$
\begin{aligned}
\langle \text{expr} \rangle \ ::= \quad & \langle \text{expr} \rangle \ \texttt{-} \ \langle \text{expr} \rangle \\
\mid \ & \langle \text{expr} \rangle \ \texttt{*} \ \langle \text{expr} \rangle \\
\mid \ & \texttt{(} \ \langle \text{expr} \rangle \ \texttt{)} \\
\mid \ & \texttt{-} \ \langle \text{expr} \rangle \\
\mid \ & \langle \text{NB} \rangle
\end{aligned}
$$

Reconstruction de l'arbre :



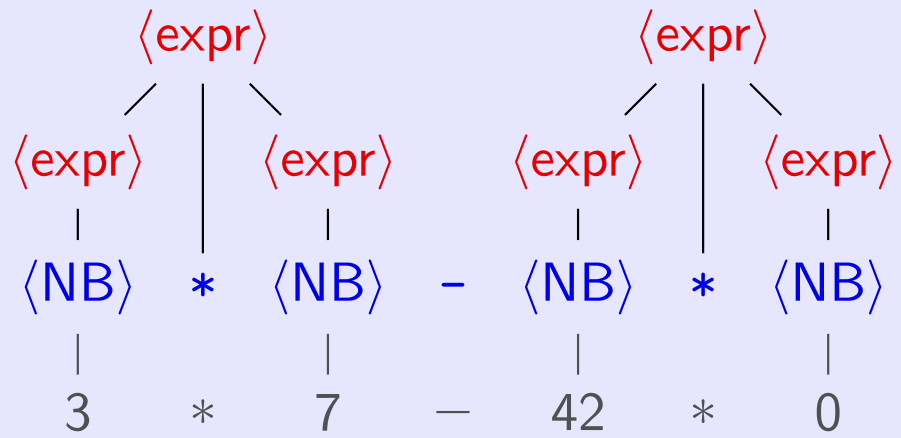
Parseur LR en action

Construction de l'abre profondeur gauche remontant

Grammaire :

$$
\begin{aligned}
\langle \text{expr} \rangle \ ::= \ & \langle \text{expr} \rangle \ \texttt{-} \ \langle \text{expr} \rangle \\
| \ & \langle \text{expr} \rangle \ \texttt{*} \ \langle \text{expr} \rangle \\
| \ & \texttt{(} \ \langle \text{expr} \rangle \ \texttt{)} \\
| \ & \texttt{-} \ \langle \text{expr} \rangle \\
| \ & \langle \text{NB} \rangle
\end{aligned}
$$

Reconstruction de l'arbre :



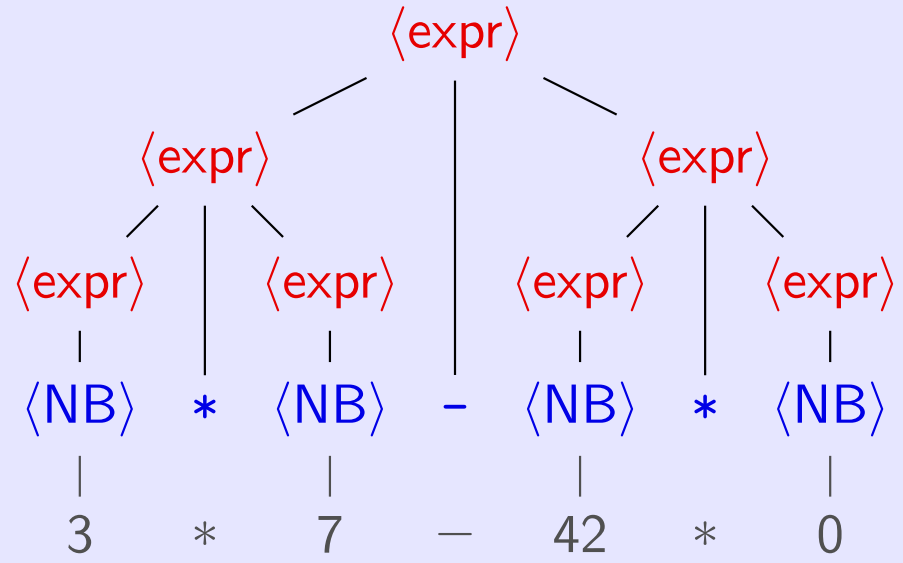
Parseur LR en action

Construction de l'abre profondeur gauche remontant

Grammaire :

$$
\begin{aligned}
\langle \text{expr} \rangle \ ::= \quad & \langle \text{expr} \rangle \ \texttt{-} \ \langle \text{expr} \rangle \\
\mid \ & \langle \text{expr} \rangle \ \texttt{*} \ \langle \text{expr} \rangle \\
\mid \ & \texttt{(} \ \langle \text{expr} \rangle \ \texttt{)} \\
\mid \ & \texttt{-} \ \langle \text{expr} \rangle \\
\mid \ & \langle \text{NB} \rangle
\end{aligned}
$$

Reconstruction de l'arbre :



Parseur LR en action

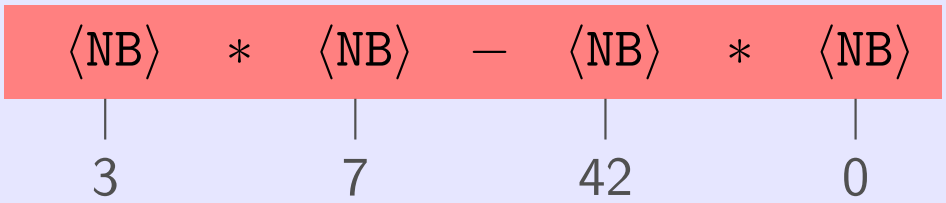
Le parseur ne lit que les noms des tokens construits

Sur un arbre on ne regarde que la racine.

Grammaire :

$$
\begin{aligned}
\langle \texttt{exp} \rangle &::= \langle \texttt{exp} \rangle - \langle \texttt{terme} \rangle \\
&\mid \langle \texttt{terme} \rangle \\
\langle \texttt{terme} \rangle &::= \langle \texttt{terme} \rangle * \langle \texttt{fac} \rangle \\
&\mid \langle \texttt{fac} \rangle \\
\langle \texttt{fac} \rangle &::= (\langle \texttt{exp} \rangle) \\
&\mid -\langle \texttt{fac} \rangle \\
&\mid \langle \texttt{NB} \rangle
\end{aligned}
$$

Reconstruction de l'arbre :



Parseur LR en action

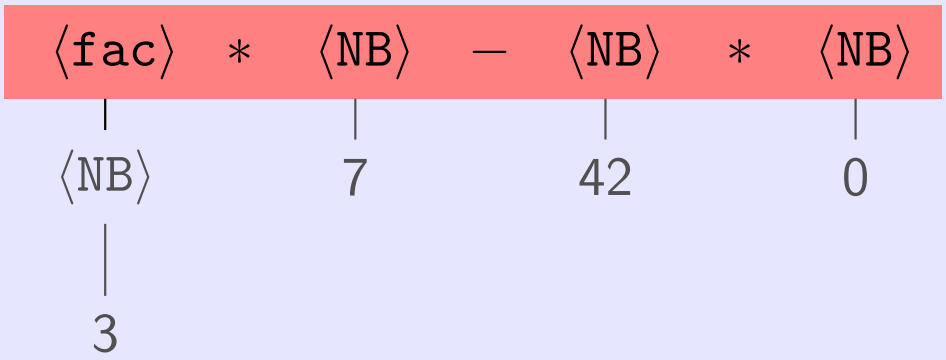
Le parseur ne lit que les noms des tokens construits

Sur un arbre on ne regarde que la racine.

Grammaire :

$$\begin{aligned}
\langle \texttt{exp} \rangle &::= \langle \texttt{exp} \rangle - \langle \texttt{terme} \rangle \\
&\mid \langle \texttt{terme} \rangle \\
\langle \texttt{terme} \rangle &::= \langle \texttt{terme} \rangle * \langle \texttt{fac} \rangle \\
&\mid \langle \texttt{fac} \rangle \\
\langle \texttt{fac} \rangle &::= (\langle \texttt{exp} \rangle) \\
&\mid -\langle \texttt{fac} \rangle \\
&\mid \langle \texttt{NB} \rangle
\end{aligned}$$

Reconstruction de l'arbre :



Parseur LR en action

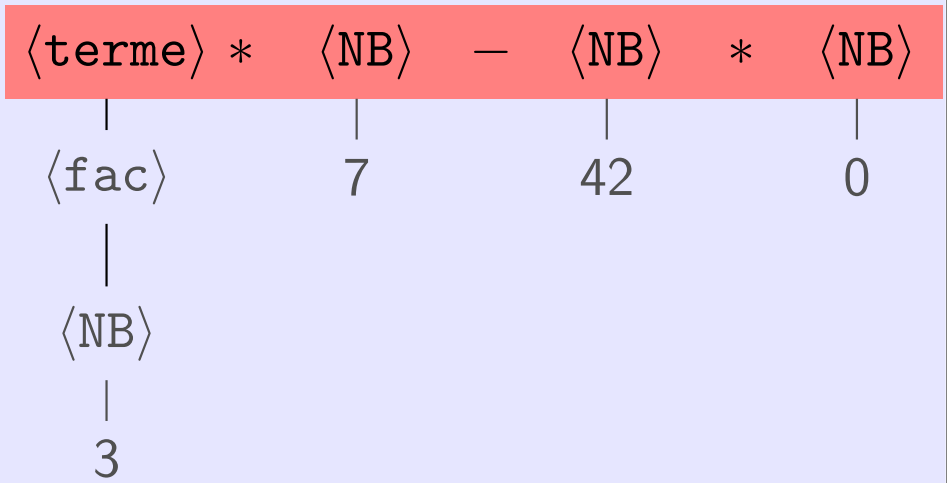
Le parseur ne lit que les noms des tokens construits

Sur un arbre on ne regarde que la racine.

Grammaire :

$$
\begin{aligned}
\langle \texttt{exp} \rangle &::= \langle \texttt{exp} \rangle - \langle \texttt{terme} \rangle \\
&\mid \langle \texttt{terme} \rangle \\
\langle \texttt{terme} \rangle &::= \langle \texttt{terme} \rangle * \langle \texttt{fac} \rangle \\
&\mid \langle \texttt{fac} \rangle \\
\langle \texttt{fac} \rangle &::= (\langle \texttt{exp} \rangle) \\
&\mid -\langle \texttt{fac} \rangle \\
&\mid \langle \texttt{NB} \rangle
\end{aligned}
$$

Reconstruction de l'arbre :



Parseur LR en action

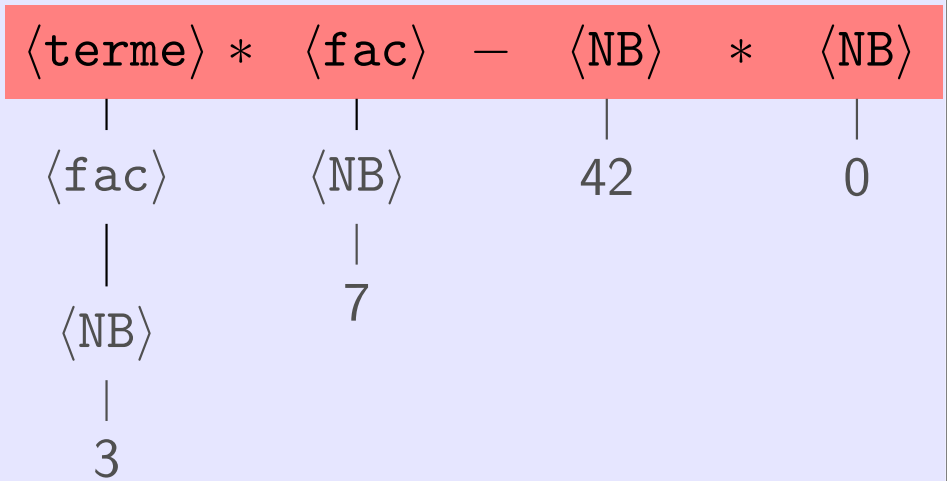
Le parseur ne lit que les noms des tokens construits

Sur un arbre on ne regarde que la racine.

Grammaire :

$$
\begin{aligned}
\langle \texttt{exp} \rangle &::= \langle \texttt{exp} \rangle - \langle \texttt{terme} \rangle \\
&\mid \langle \texttt{terme} \rangle \\
\langle \texttt{terme} \rangle &::= \langle \texttt{terme} \rangle * \langle \texttt{fac} \rangle \\
&\mid \langle \texttt{fac} \rangle \\
\langle \texttt{fac} \rangle &::= (\langle \texttt{exp} \rangle) \\
&\mid -\langle \texttt{fac} \rangle \\
&\mid \langle \texttt{NB} \rangle
\end{aligned}
$$

Reconstruction de l'arbre :



Parseur LR en action

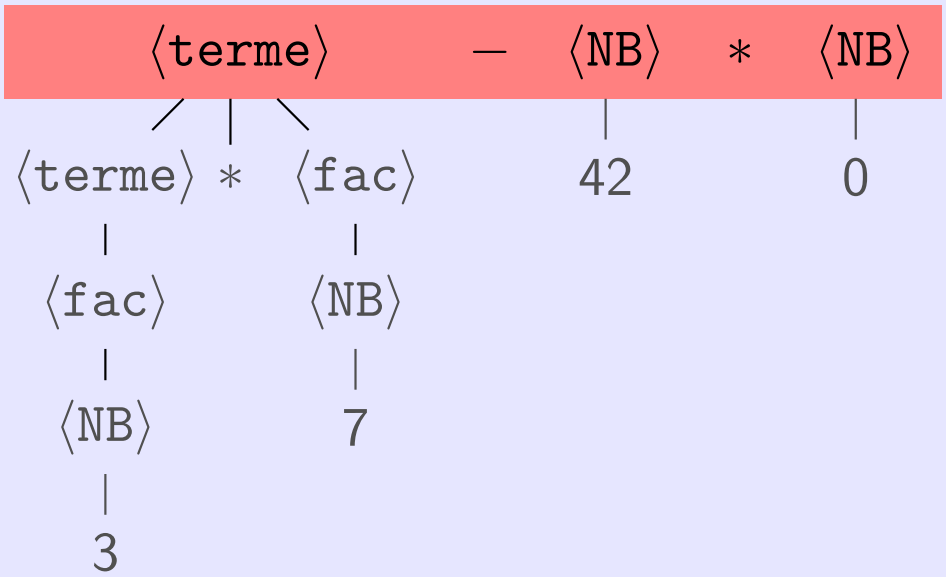
Le parseur ne lit que les noms des tokens construits

Sur un arbre on ne regarde que la racine.

Grammaire :

$$
\begin{aligned}
\langle \texttt{exp} \rangle &::= \langle \texttt{exp} \rangle - \langle \texttt{terme} \rangle \\
&\mid \langle \texttt{terme} \rangle \\
\langle \texttt{terme} \rangle &::= \langle \texttt{terme} \rangle * \langle \texttt{fac} \rangle \\
&\mid \langle \texttt{fac} \rangle \\
\langle \texttt{fac} \rangle &::= (\langle \texttt{exp} \rangle) \\
&\mid -\langle \texttt{fac} \rangle \\
&\mid \langle \texttt{NB} \rangle
\end{aligned}
$$

Reconstruction de l'arbre :



Parseur LR en action

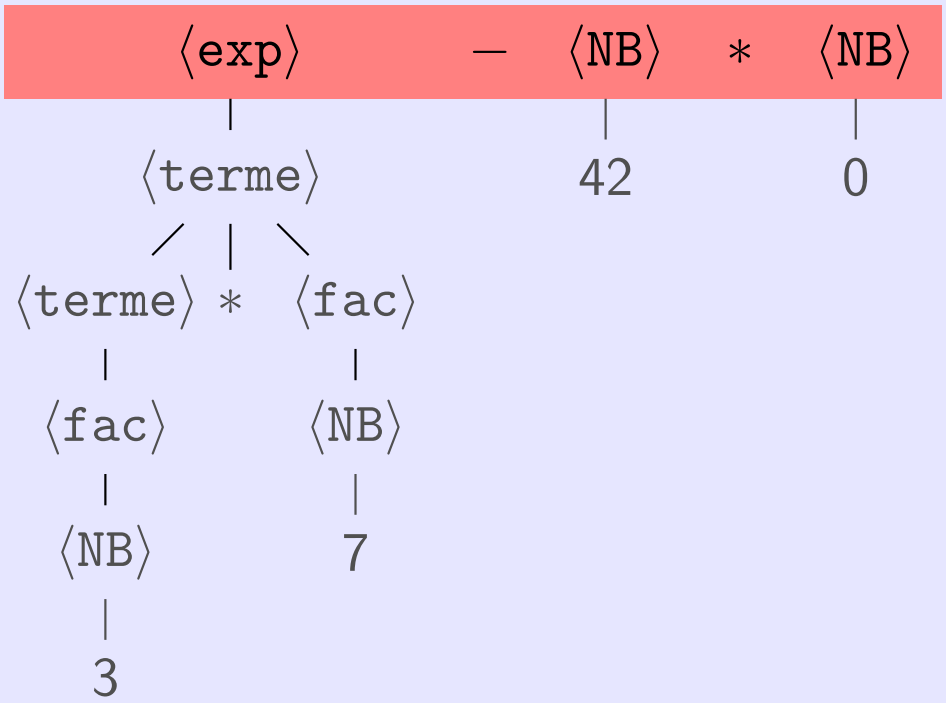
Le parseur ne lit que les noms des tokens construits

Sur un arbre on ne regarde que la racine.

Grammaire :

$$
\begin{aligned}
\langle \texttt{exp} \rangle &::= \langle \texttt{exp} \rangle - \langle \texttt{terme} \rangle \\
&\mid \langle \texttt{terme} \rangle \\
\langle \texttt{terme} \rangle &::= \langle \texttt{terme} \rangle * \langle \texttt{fac} \rangle \\
&\mid \langle \texttt{fac} \rangle \\
\langle \texttt{fac} \rangle &::= (\langle \texttt{exp} \rangle) \\
&\mid -\langle \texttt{fac} \rangle \\
&\mid \langle \texttt{NB} \rangle
\end{aligned}
$$

Reconstruction de l'arbre :



Parseur LR en action

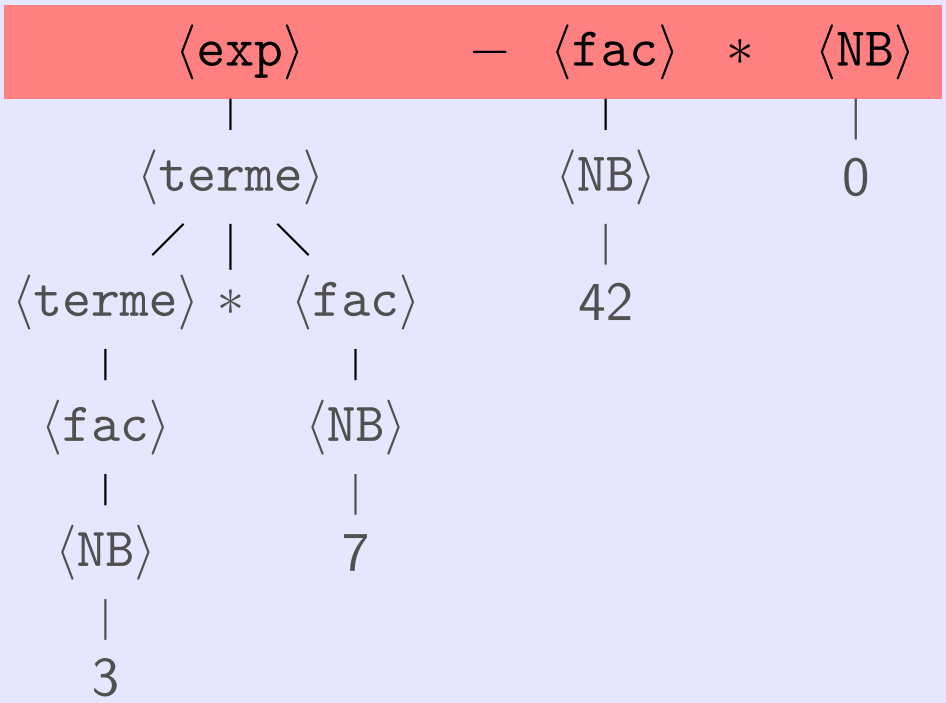
Le parseur ne lit que les noms des tokens construits

Sur un arbre on ne regarde que la racine.

Grammaire :

$$
\begin{aligned}
\langle \texttt{exp} \rangle &::= \langle \texttt{exp} \rangle - \langle \texttt{terme} \rangle \\
&\mid \langle \texttt{terme} \rangle \\
\langle \texttt{terme} \rangle &::= \langle \texttt{terme} \rangle * \langle \texttt{fac} \rangle \\
&\mid \langle \texttt{fac} \rangle \\
\langle \texttt{fac} \rangle &::= (\langle \texttt{exp} \rangle) \\
&\mid -\langle \texttt{fac} \rangle \\
&\mid \langle \texttt{NB} \rangle
\end{aligned}
$$

Reconstruction de l'arbre :



Parseur LR en action

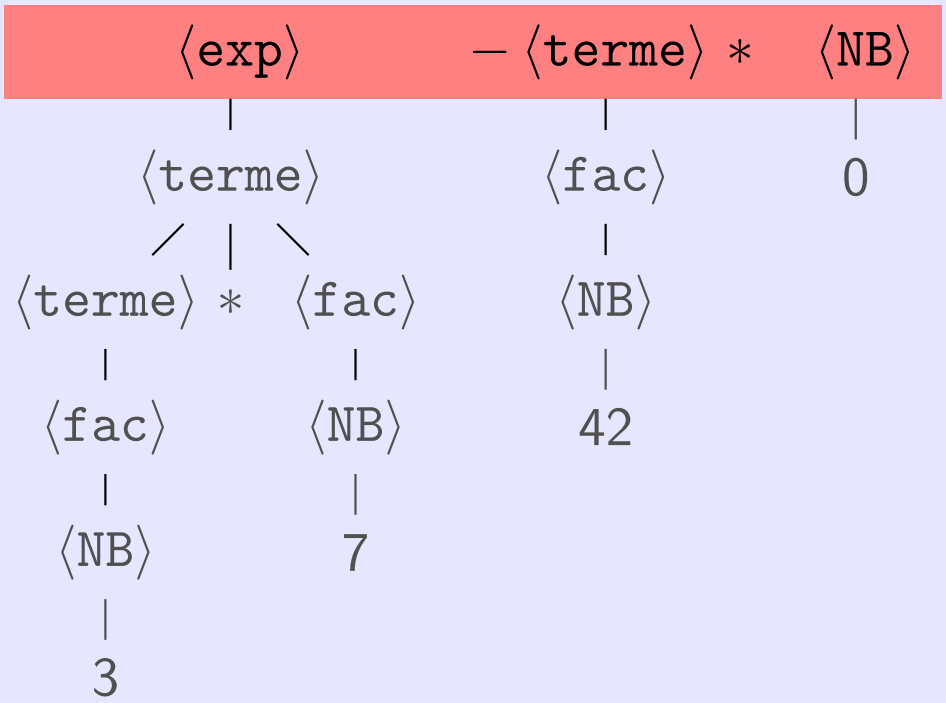
Le parseur ne lit que les noms des tokens construits

Sur un arbre on ne regarde que la racine.

Grammaire :

$$
\begin{aligned}
\langle \texttt{exp} \rangle &::= \langle \texttt{exp} \rangle - \langle \texttt{terme} \rangle \\
&\mid \langle \texttt{terme} \rangle \\
\langle \texttt{terme} \rangle &::= \langle \texttt{terme} \rangle * \langle \texttt{fac} \rangle \\
&\mid \langle \texttt{fac} \rangle \\
\langle \texttt{fac} \rangle &::= (\langle \texttt{exp} \rangle) \\
&\mid -\langle \texttt{fac} \rangle \\
&\mid \langle \texttt{NB} \rangle
\end{aligned}
$$

Reconstruction de l'arbre :



Parseur LR en action

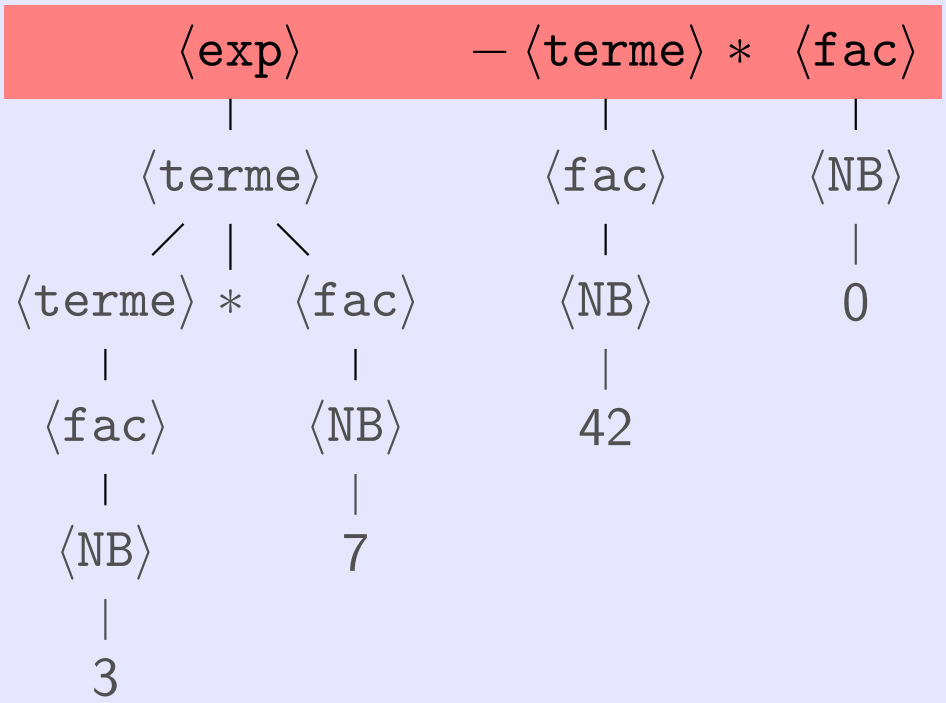
Le parseur ne lit que les noms des tokens construits

Sur un arbre on ne regarde que la racine.

Grammaire :

$$
\begin{aligned}
\langle \texttt{exp} \rangle &::= \langle \texttt{exp} \rangle - \langle \texttt{terme} \rangle \\
&\mid \langle \texttt{terme} \rangle \\
\langle \texttt{terme} \rangle &::= \langle \texttt{terme} \rangle * \langle \texttt{fac} \rangle \\
&\mid \langle \texttt{fac} \rangle \\
\langle \texttt{fac} \rangle &::= (\langle \texttt{exp} \rangle) \\
&\mid -\langle \texttt{fac} \rangle \\
&\mid \langle \texttt{NB} \rangle
\end{aligned}
$$

Reconstruction de l'arbre :



Parseur LR en action

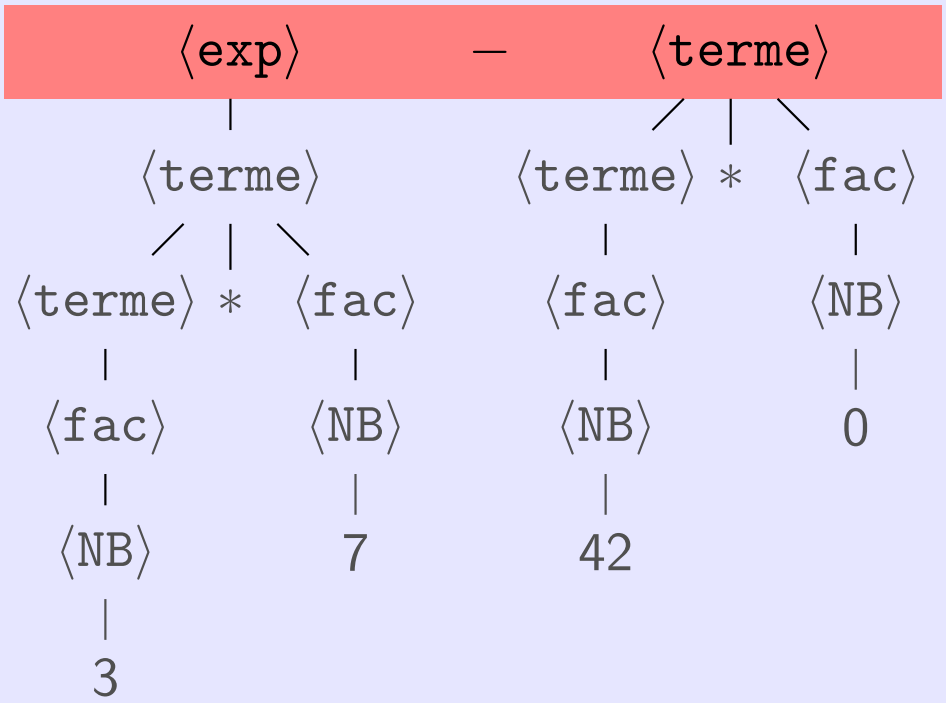
Le parseur ne lit que les noms des tokens construits

Sur un arbre on ne regarde que la racine.

Grammaire :

$$
\begin{aligned}
\langle \texttt{exp} \rangle &::= \langle \texttt{exp} \rangle - \langle \texttt{terme} \rangle \\
&\mid \langle \texttt{terme} \rangle \\
\langle \texttt{terme} \rangle &::= \langle \texttt{terme} \rangle * \langle \texttt{fac} \rangle \\
&\mid \langle \texttt{fac} \rangle \\
\langle \texttt{fac} \rangle &::= (\langle \texttt{exp} \rangle) \\
&\mid -\langle \texttt{fac} \rangle \\
&\mid \langle \texttt{NB} \rangle
\end{aligned}
$$

Reconstruction de l'arbre :



Parseur LR en action

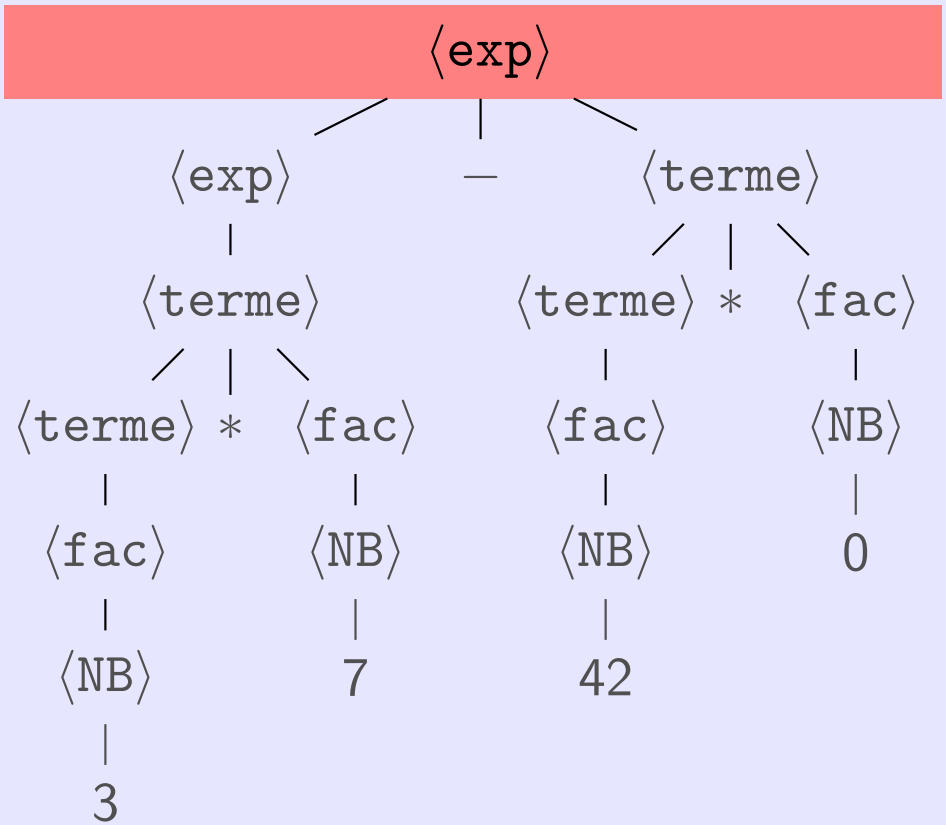
Le parseur ne lit que les noms des tokens construits

Sur un arbre on ne regarde que la racine.

Grammaire :

$$
\begin{aligned}
\langle \texttt{exp} \rangle &::= \langle \texttt{exp} \rangle - \langle \texttt{terme} \rangle \\
&\mid \langle \texttt{terme} \rangle \\
\langle \texttt{terme} \rangle &::= \langle \texttt{terme} \rangle * \langle \texttt{fac} \rangle \\
&\mid \langle \texttt{fac} \rangle \\
\langle \texttt{fac} \rangle &::= (\langle \texttt{exp} \rangle) \\
&\mid -\langle \texttt{fac} \rangle \\
&\mid \langle \texttt{NB} \rangle
\end{aligned}
$$

Reconstruction de l'arbre :



Principe du parseur non déterministe

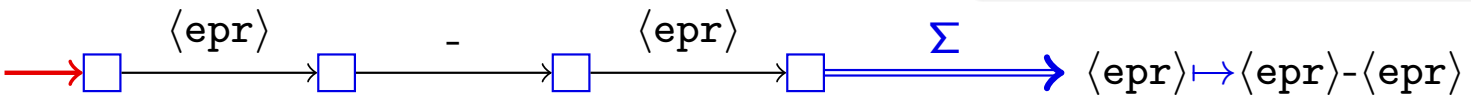
Idée : écrire un automate shift-action non déterministe

Shifts :

- lisent un mot de terminaux et non terminaux
- y cherchent le pattern d'une rêgle.

Action :

- applique la rêgle trouvée,
- forme un noeud,
- relance l'automate au début.



relancer au début

Remarque : Σ remplace `Char`

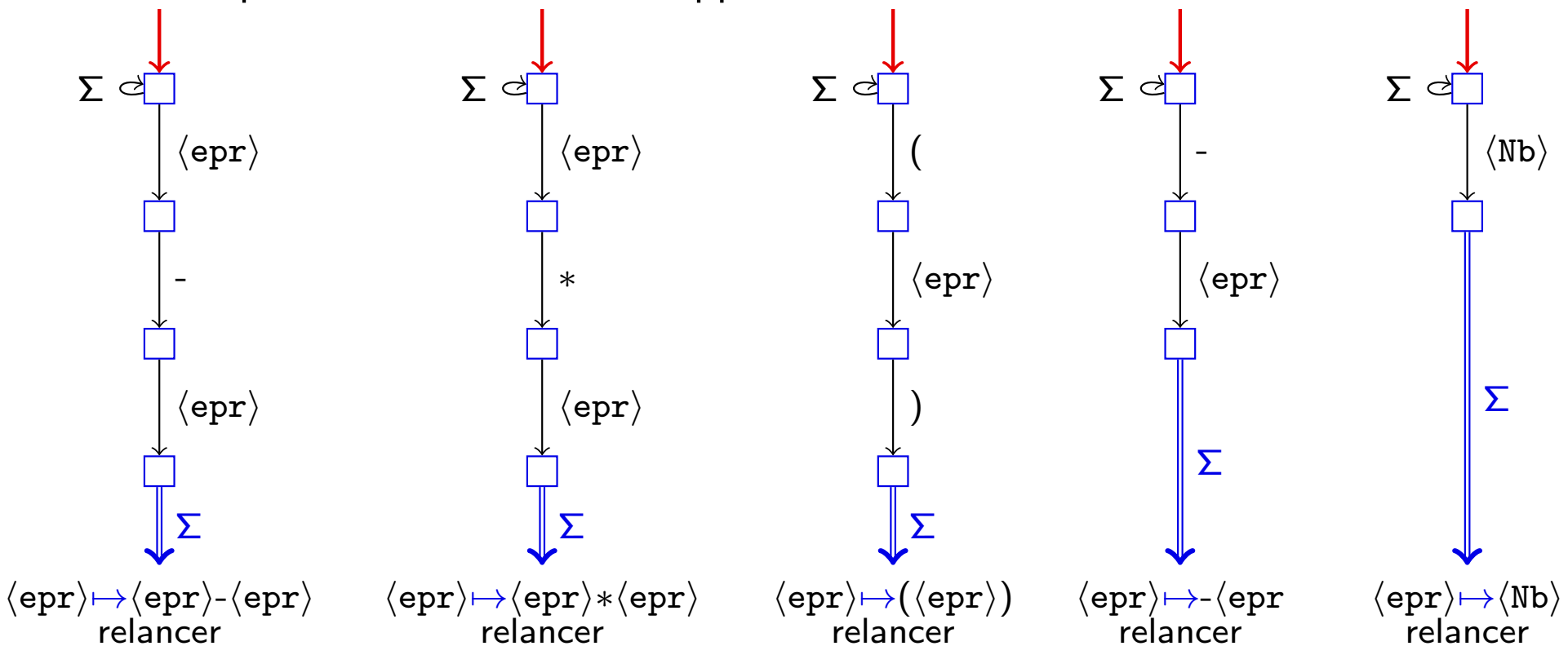
$\Sigma = \mathcal{T} \cup \mathcal{N}$ est l'ensemble des terminaux + non-terminaux

Première tentative

Idée : Faire comme le lexeur

- un petit automate pour chaque rêgle qui “test” le pattern,
- précédé d'une boucle “retardatrice” infinie,
- puis on leur met une action et on fait la somme.

Remarque : une telle action s'appel une action de réduction

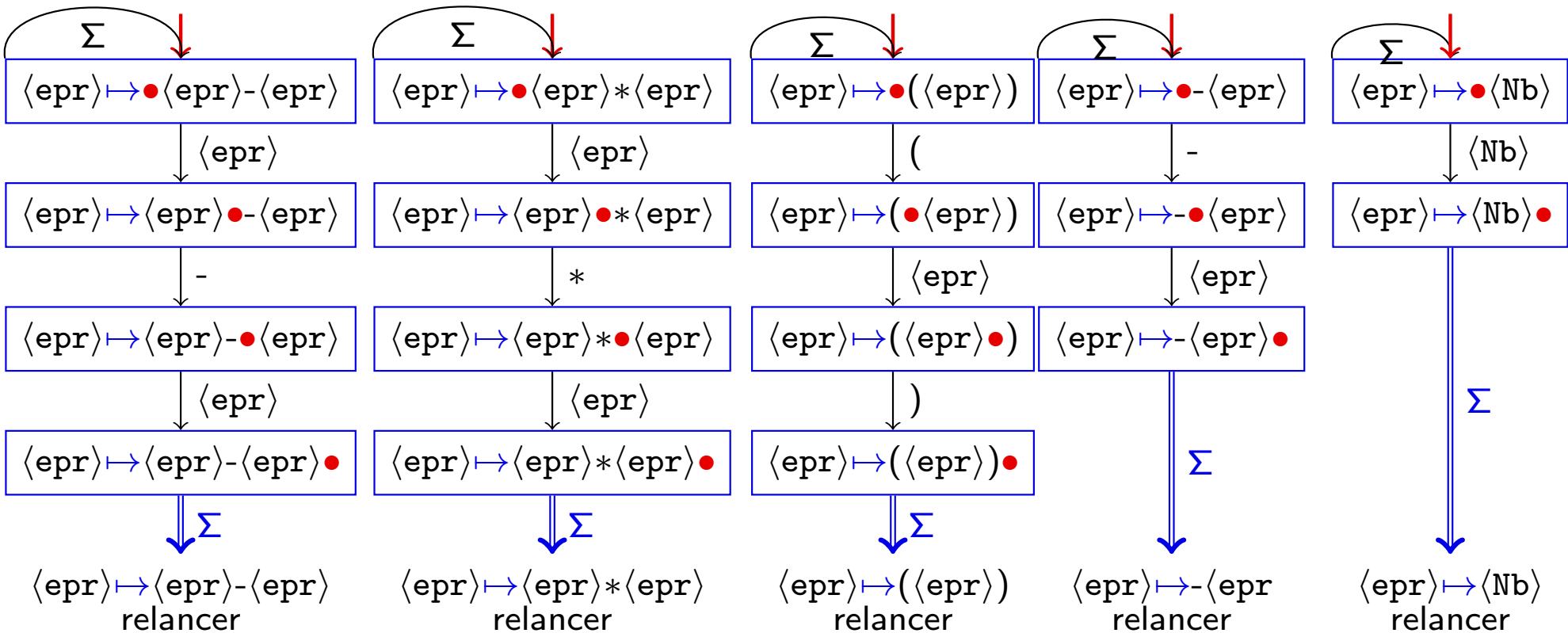


Première tentative

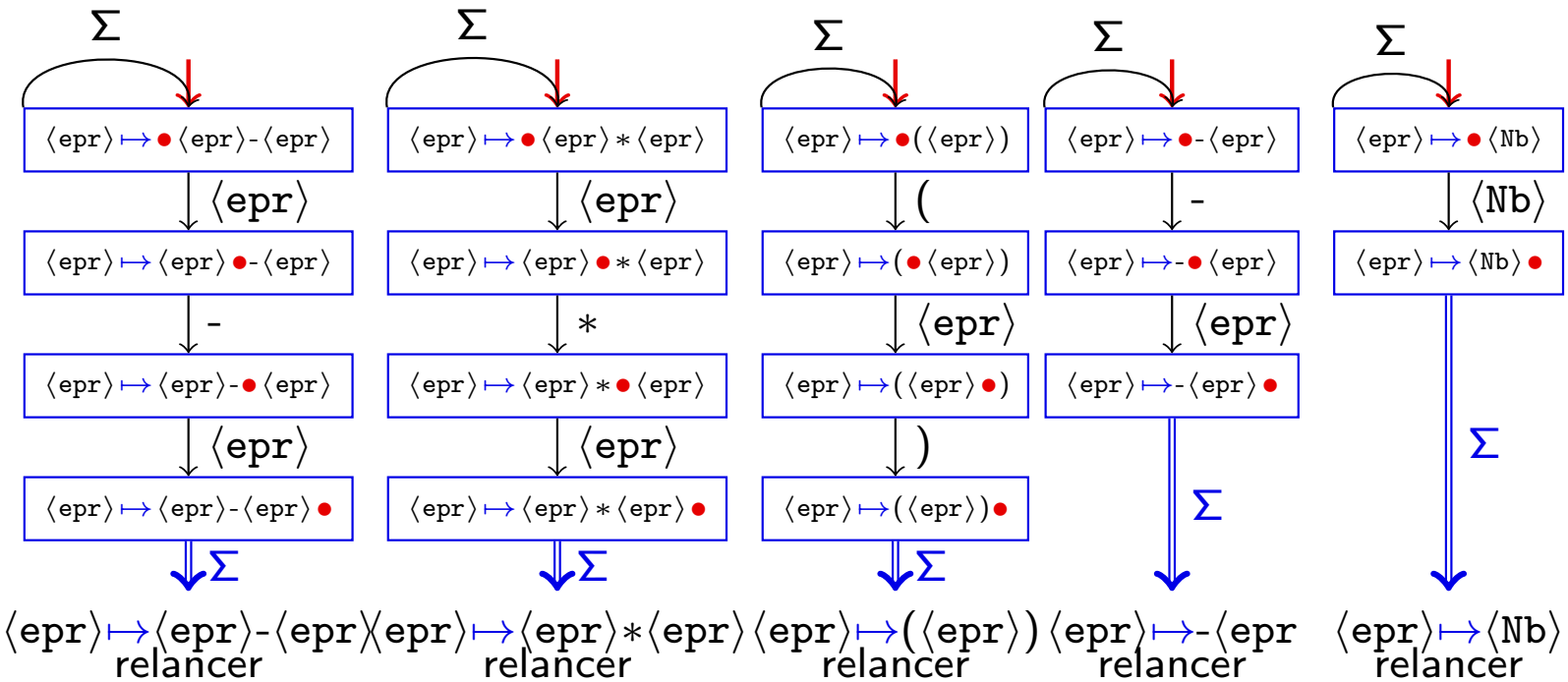
Idée : Faire comme le lexeur

- un petit automate pour chaque rêgle qui “test” le pattern,
- précédé d'une boucle “retardatrice” infinie,
- puis on leur met une action et on fait la somme.

Remarque : une telle action s'appel une action de réduction



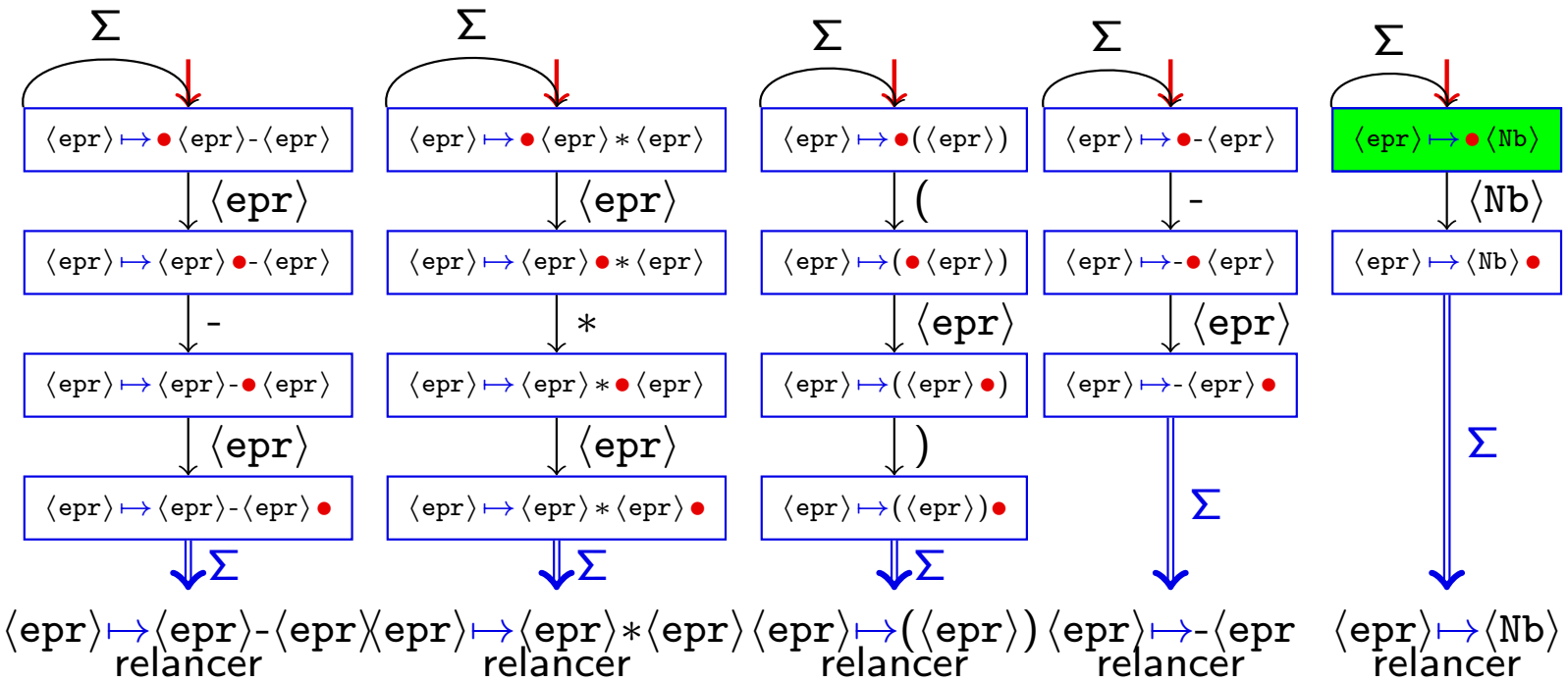
Automate du parseur LR en action

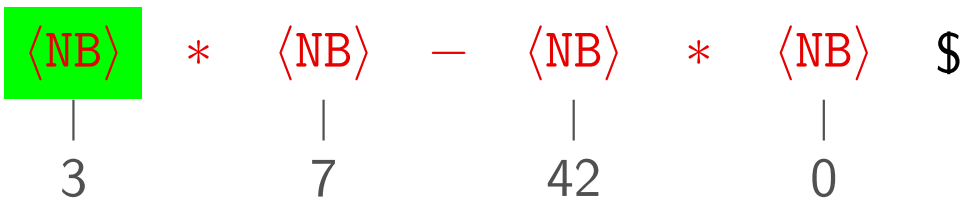


⟨NB⟩ * ⟨NB⟩ − ⟨NB⟩ * ⟨NB⟩ $

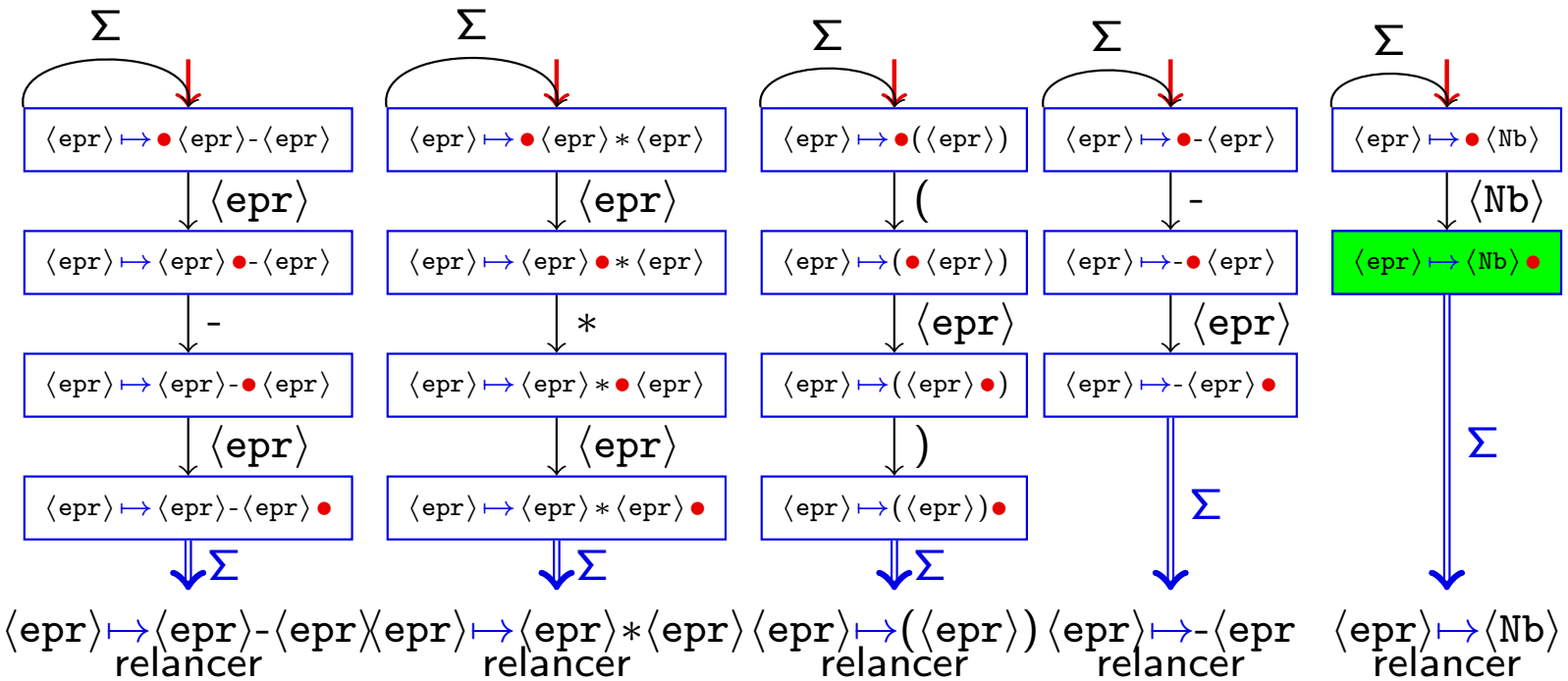
3 7 42 0

Automate du parseur LR en action





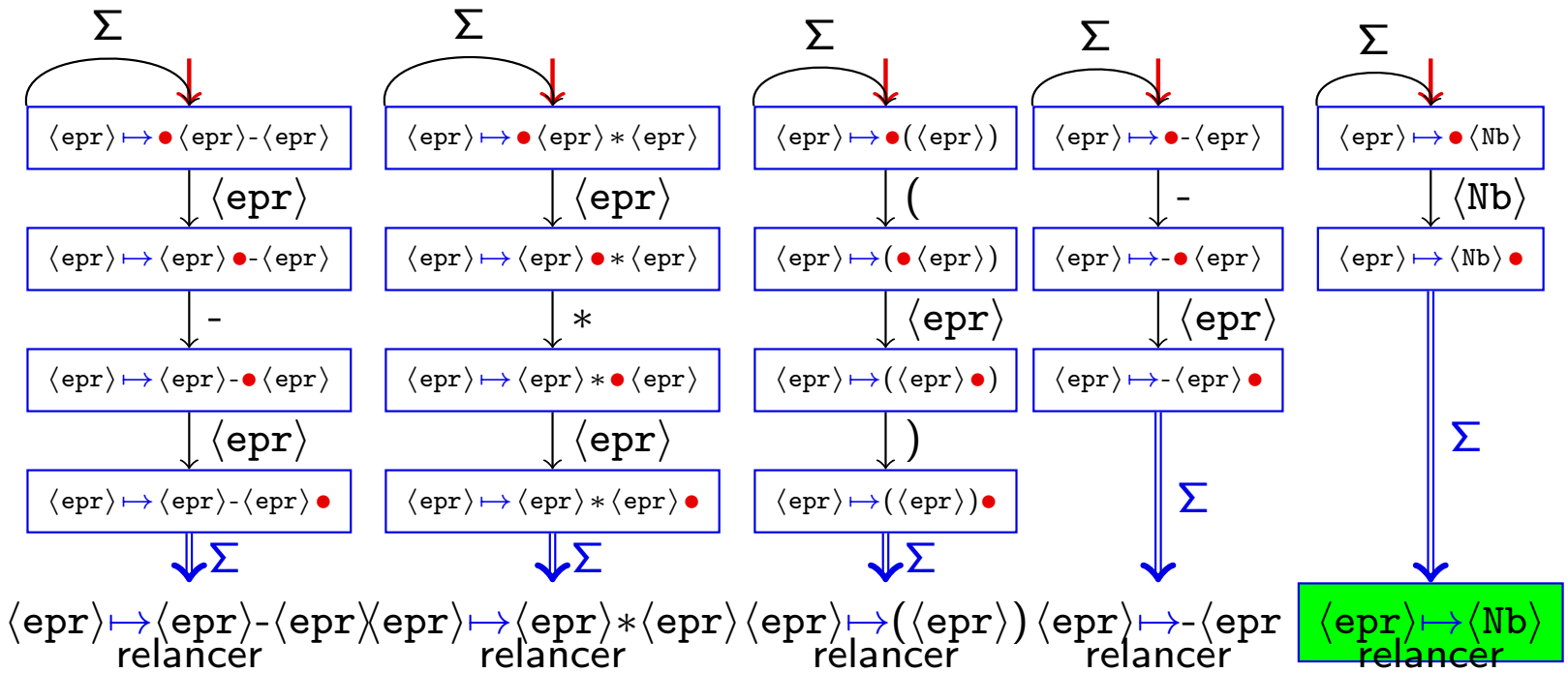
Automate du parseur LR en action

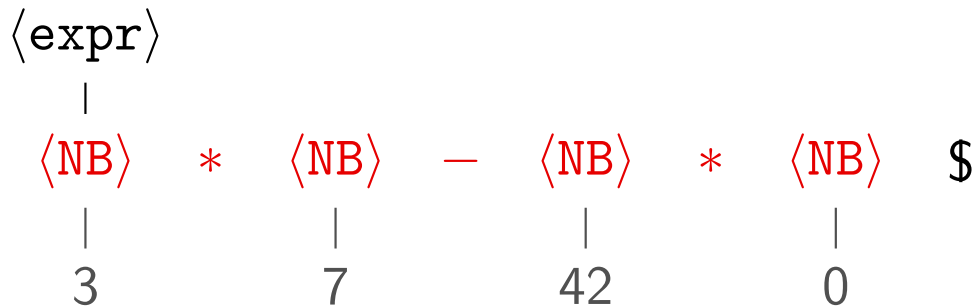


⟨NB⟩ * ⟨NB⟩ − ⟨NB⟩ * ⟨NB⟩ $

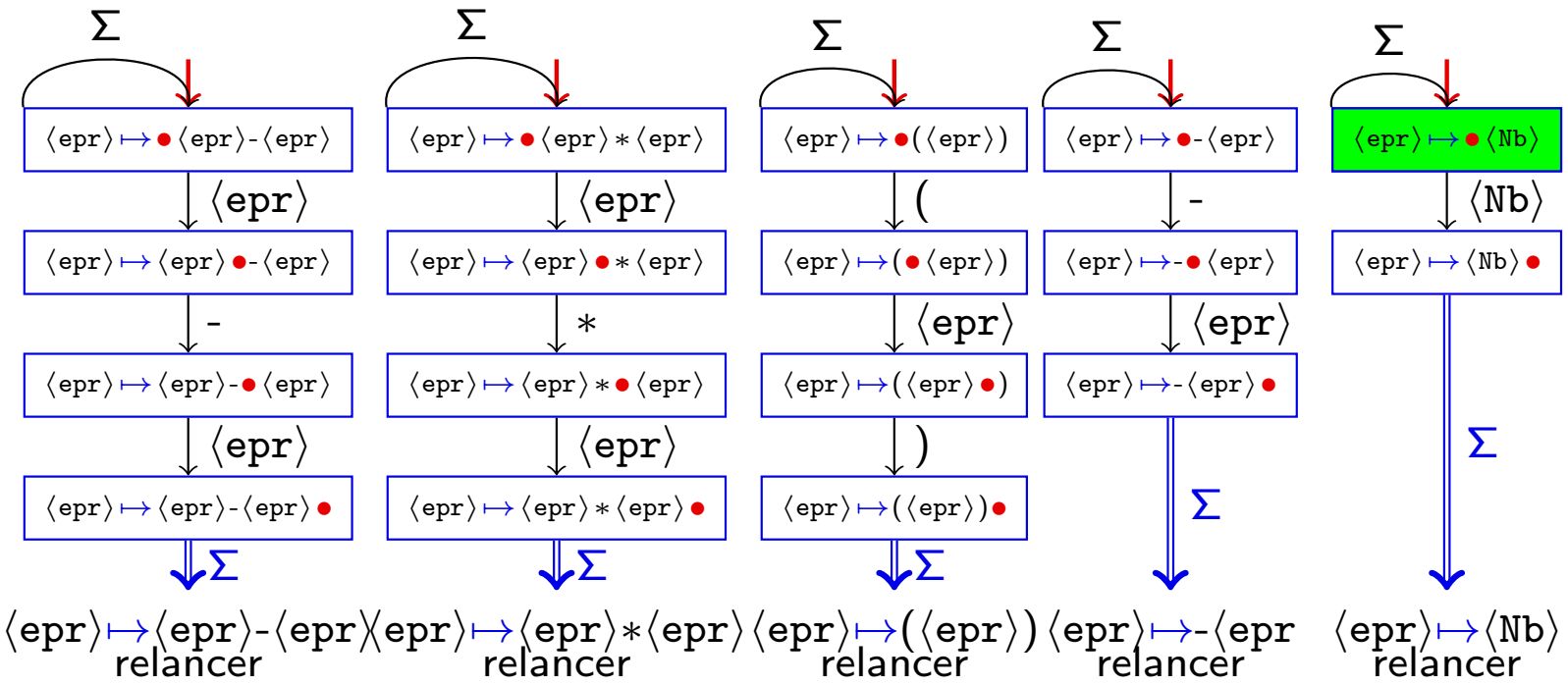
3 7 42 0

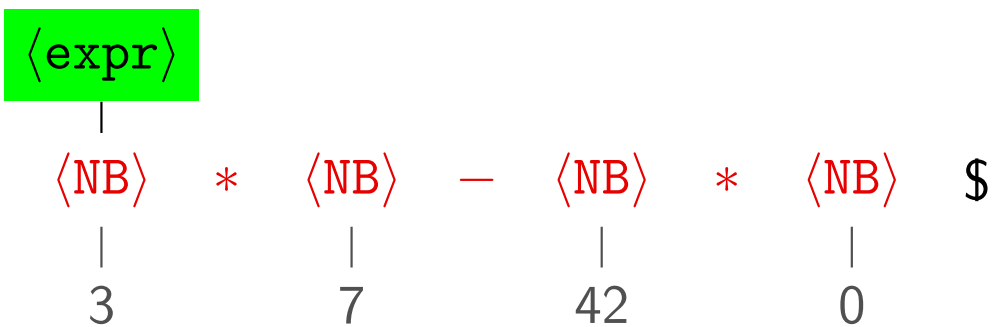
Automate du parseur LR en action



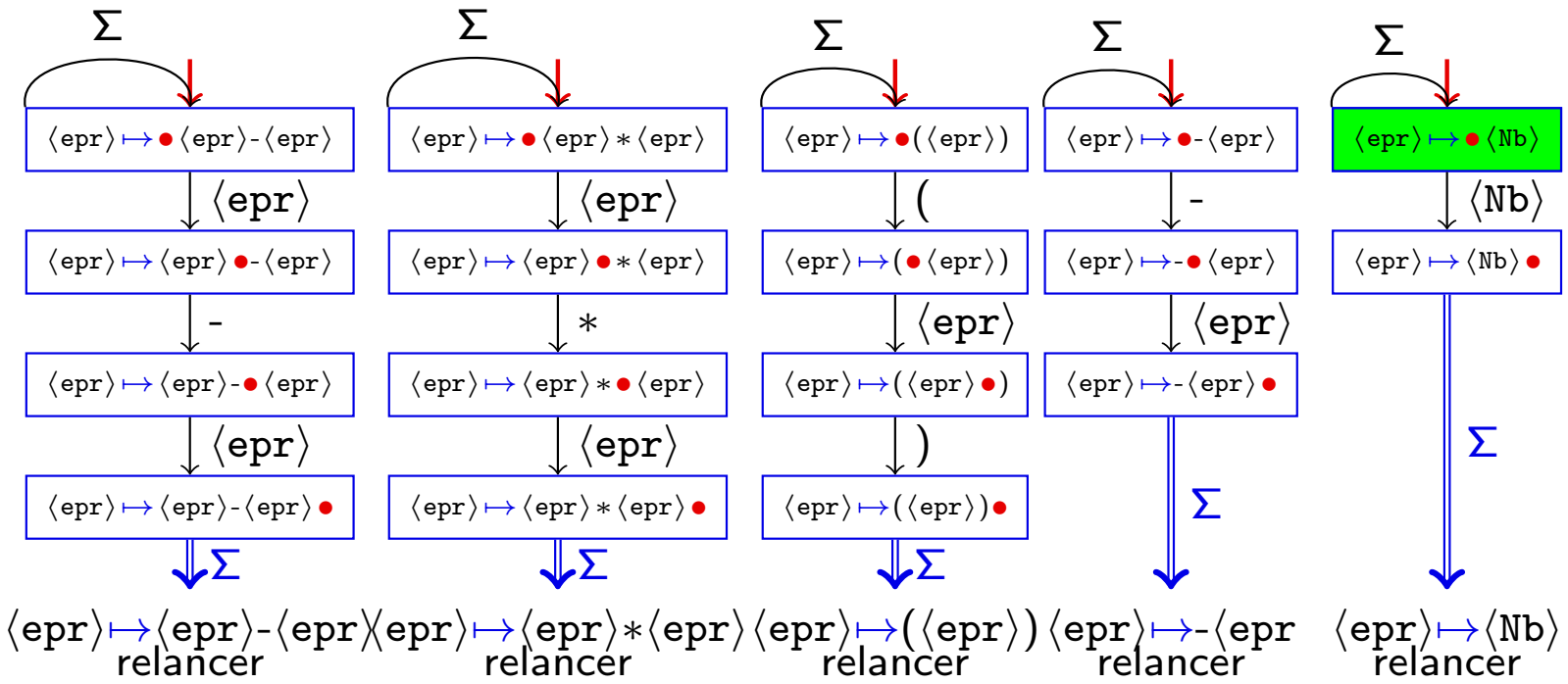


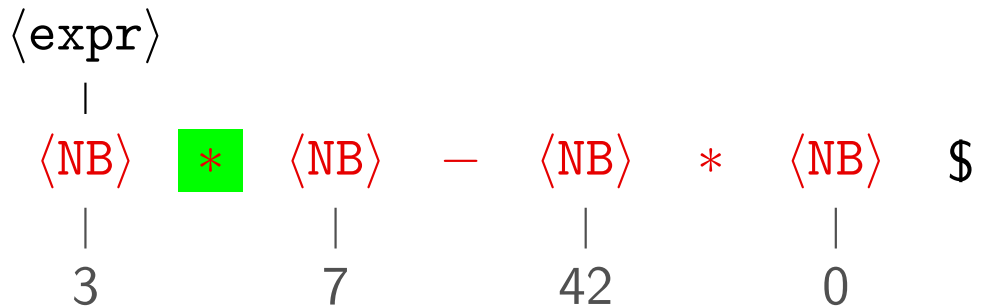
Automate du parseur LR en action



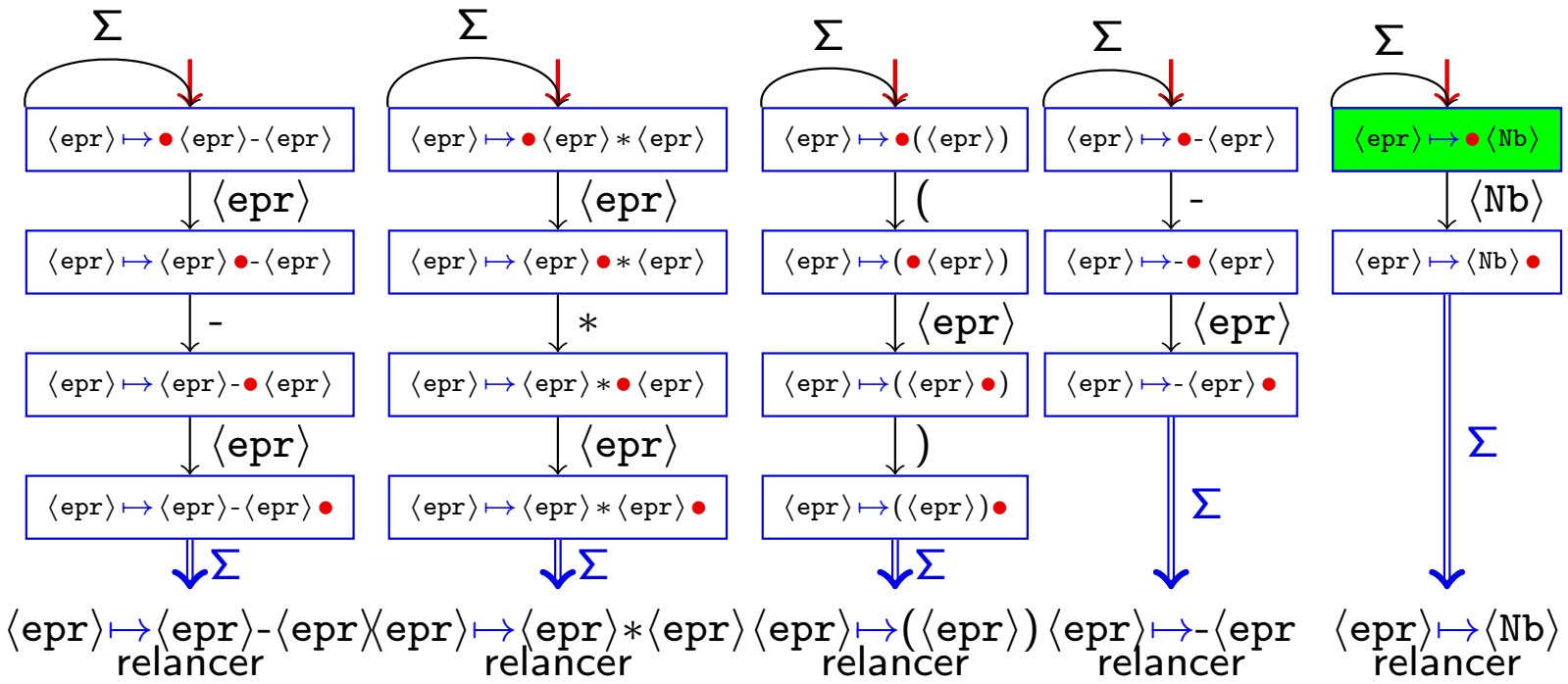


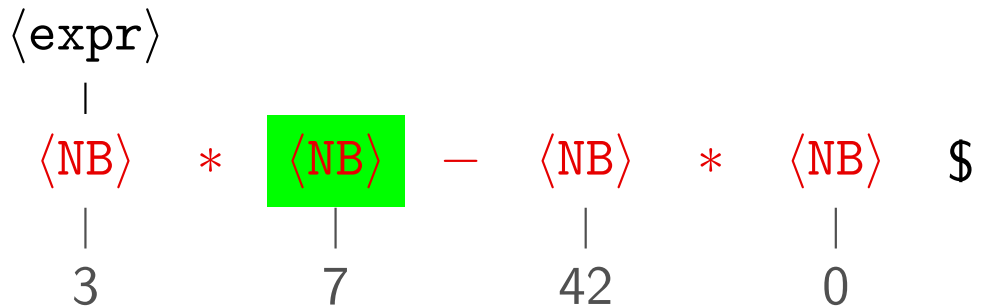
Automate du parseur LR en action



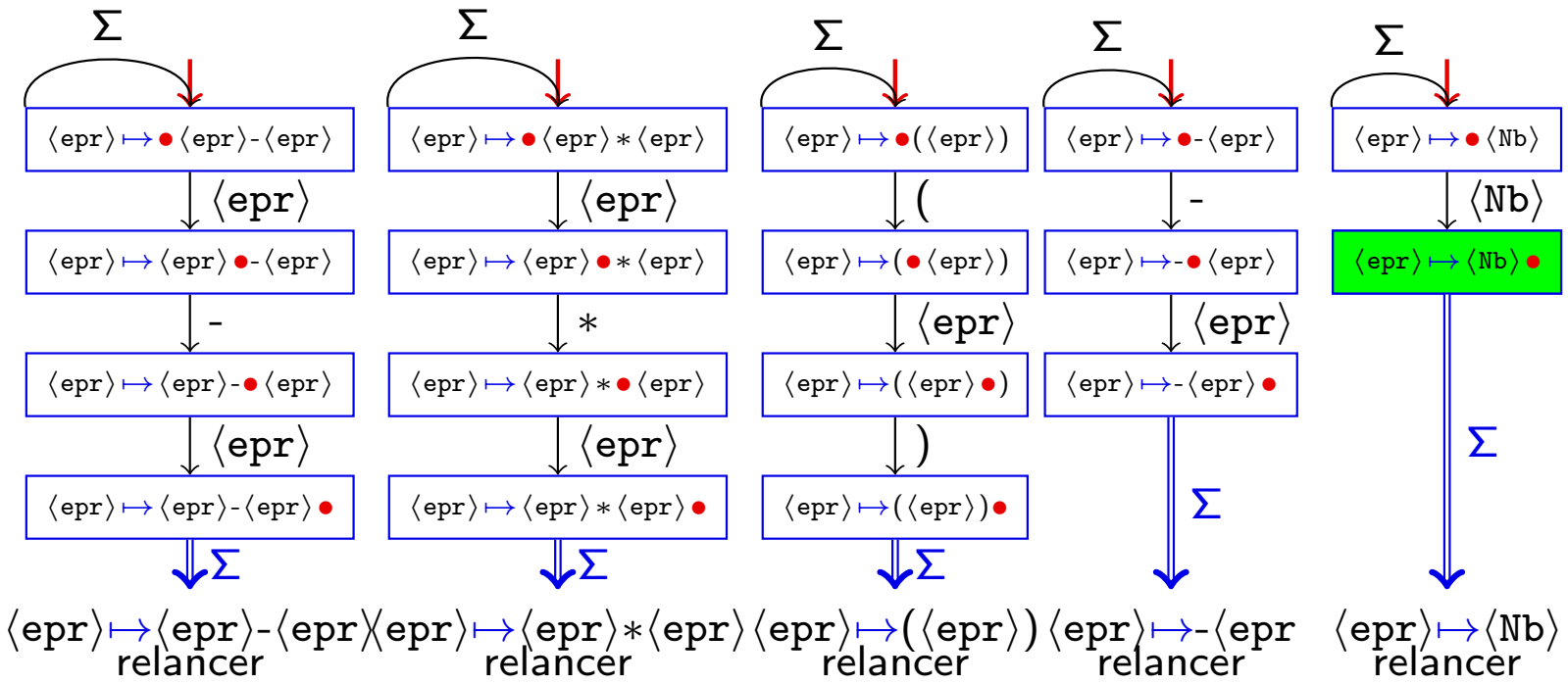


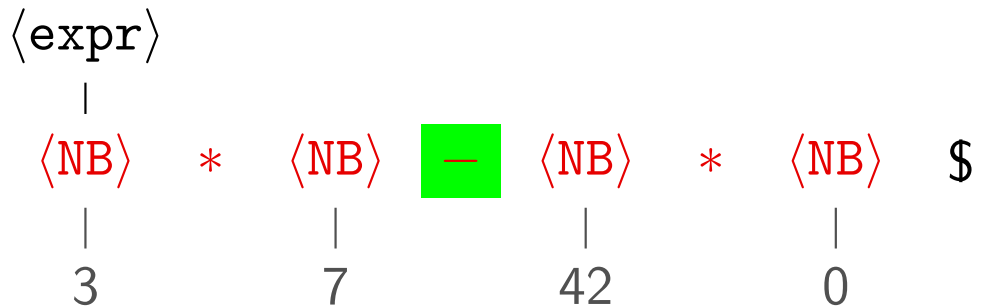
Automate du parseur LR en action



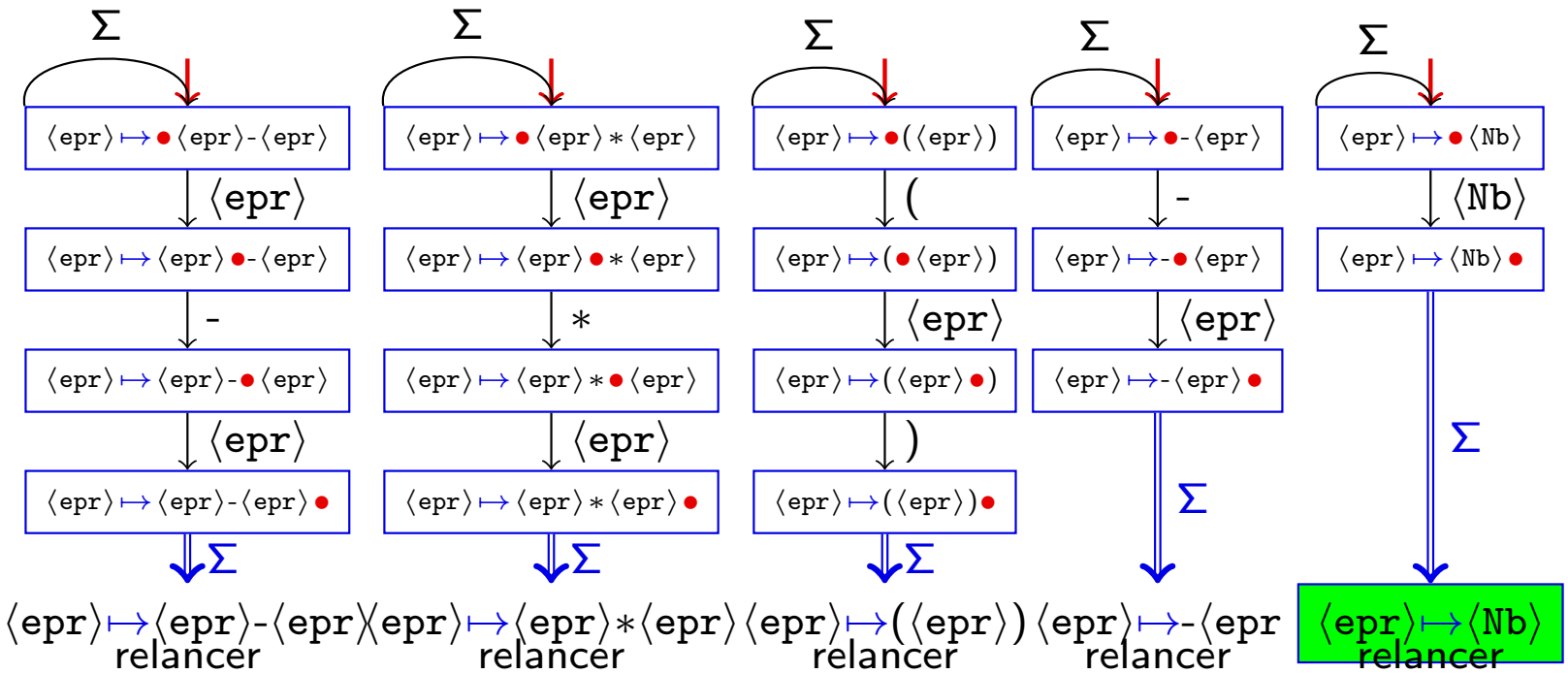


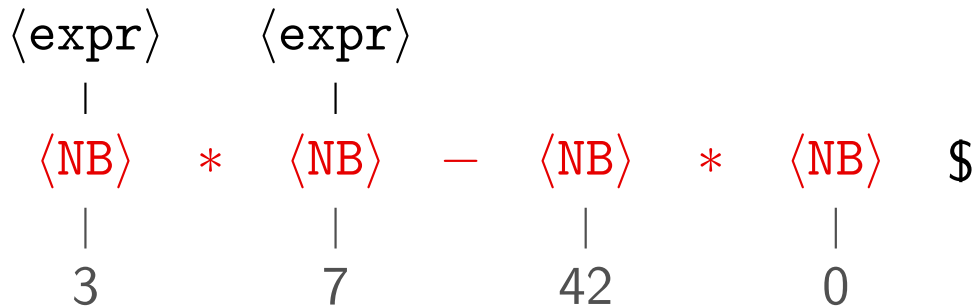
Automate du parseur LR en action



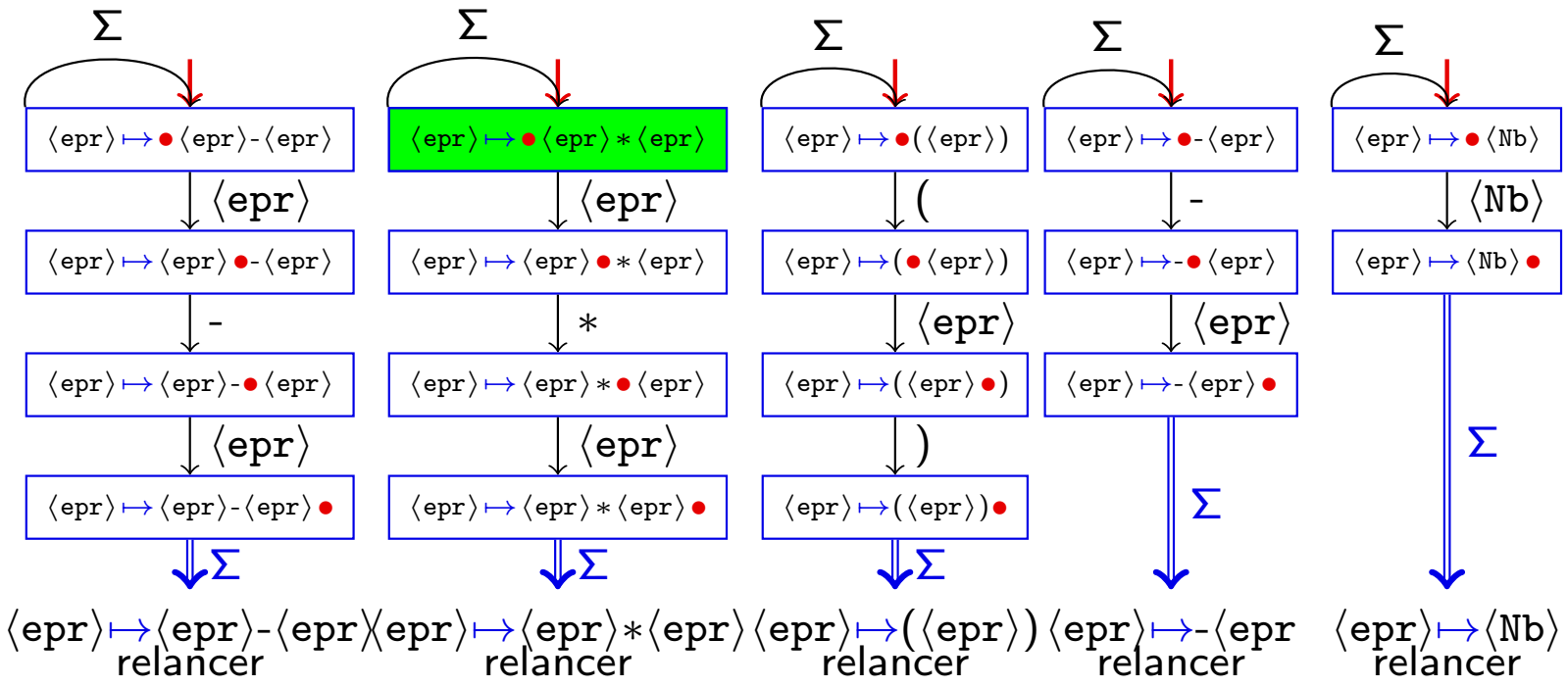


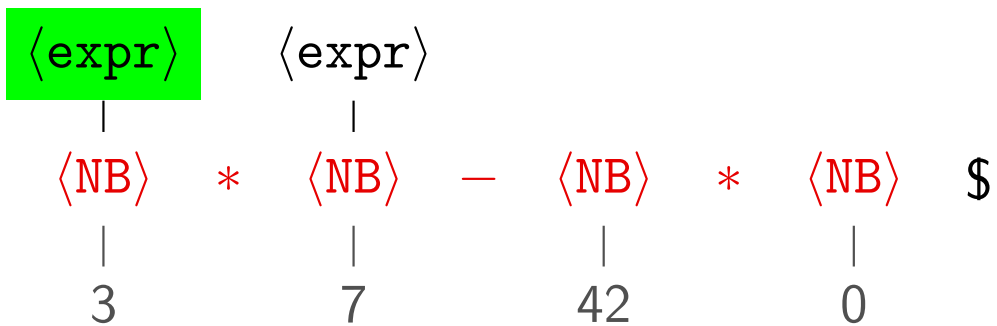
Automate du parseur LR en action



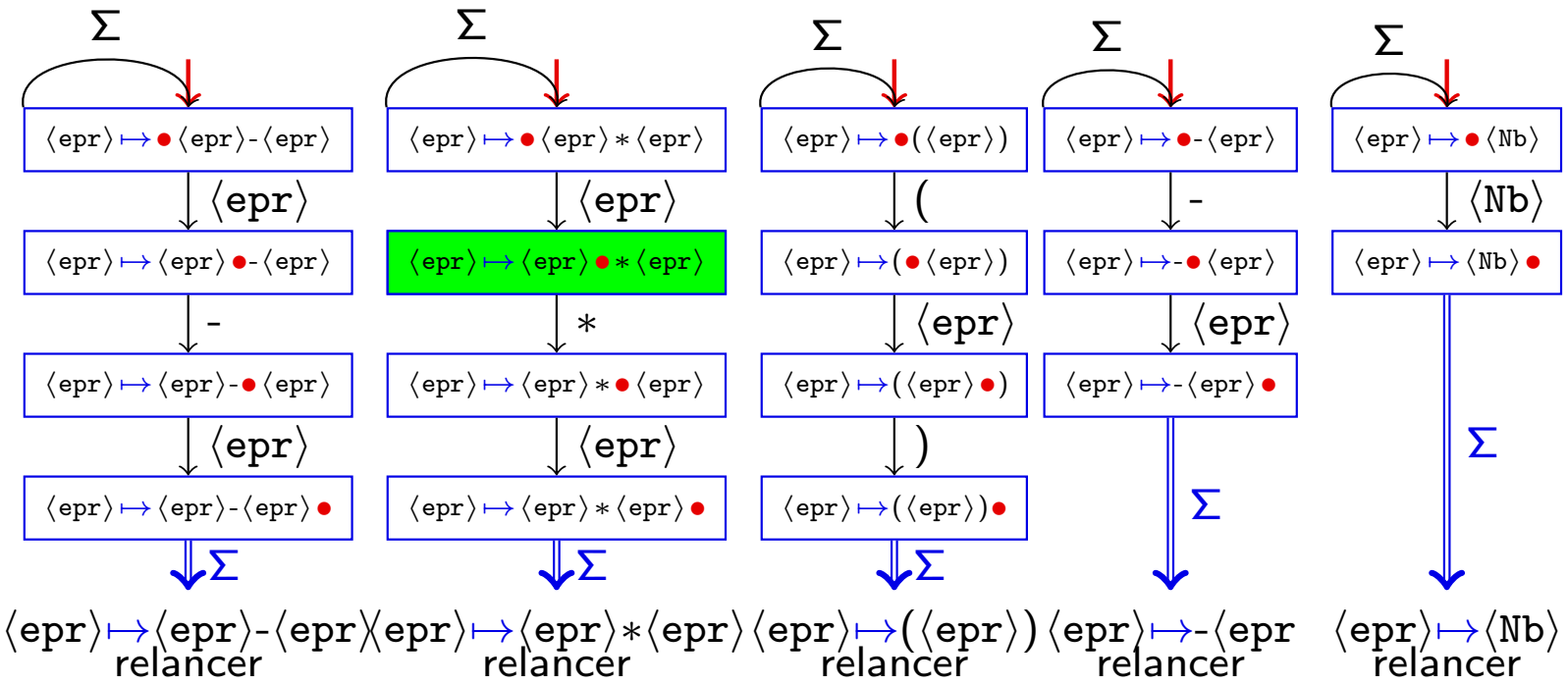


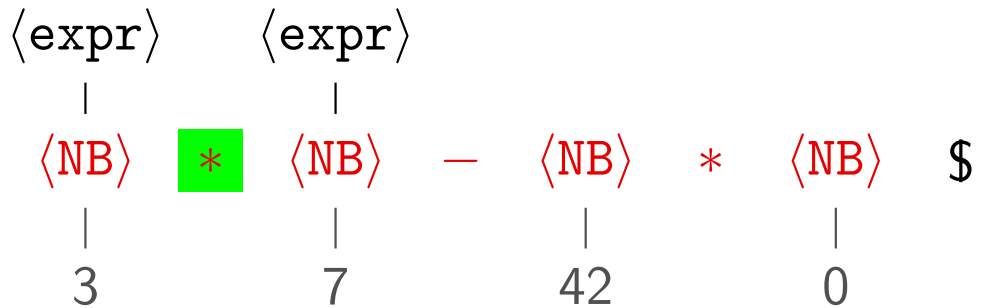
Automate du parseur LR en action



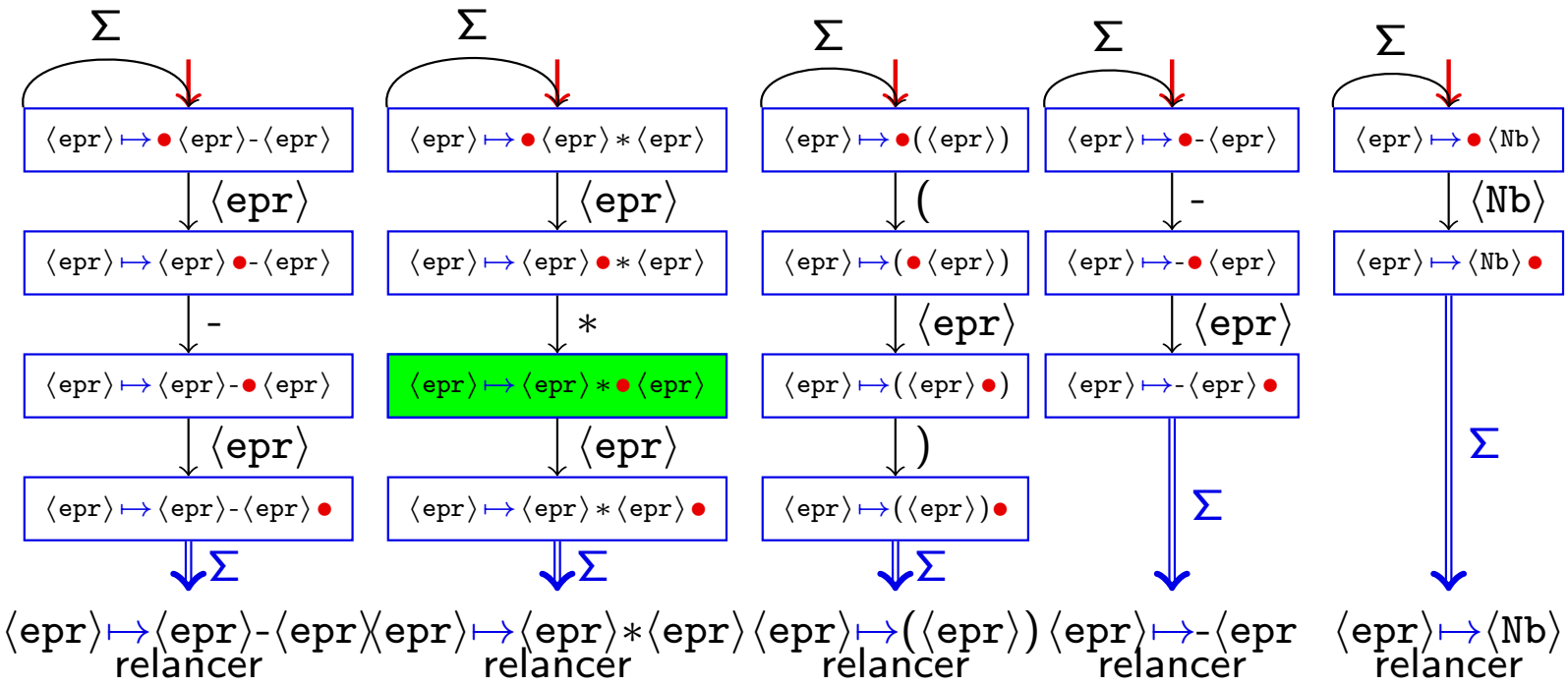


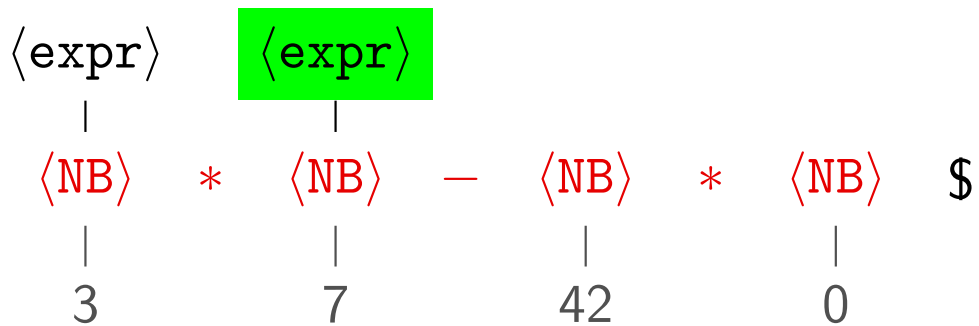
Automate du parseur LR en action



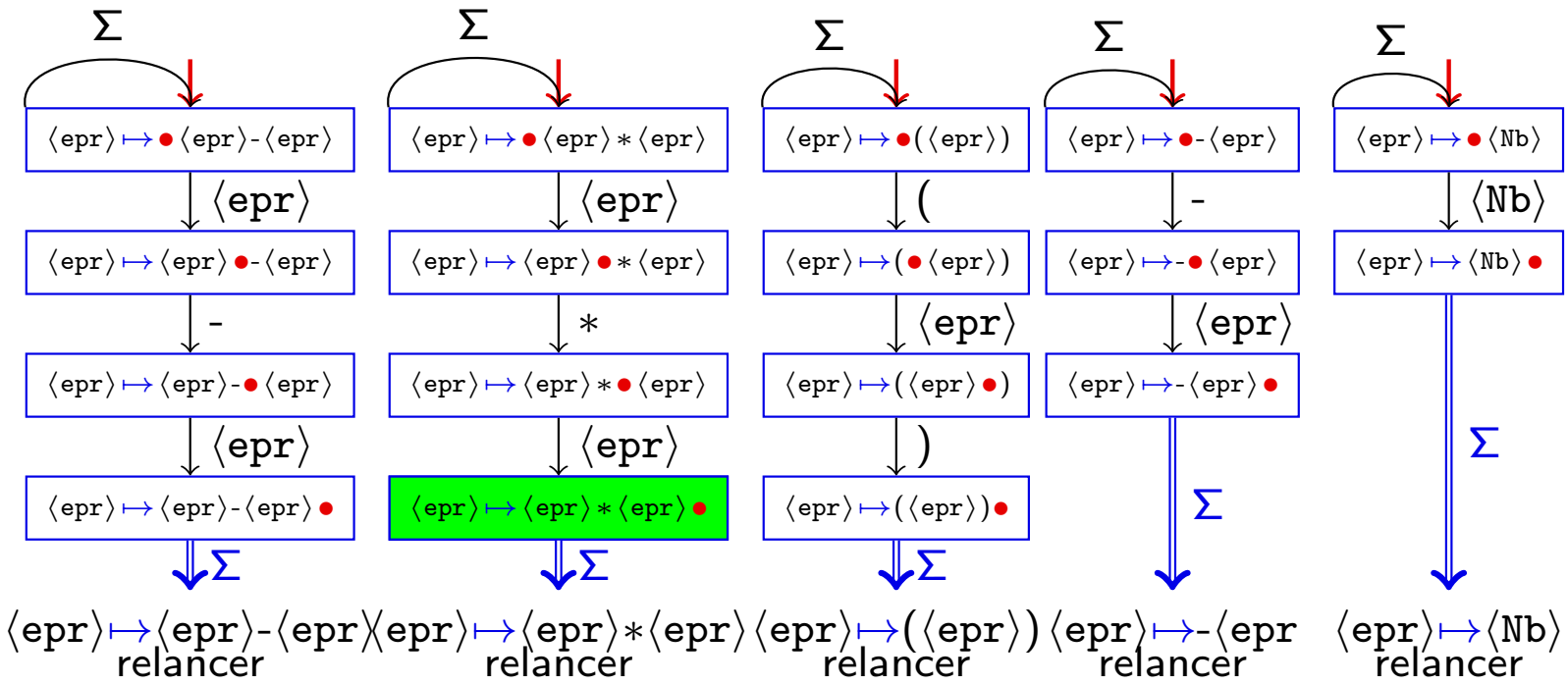


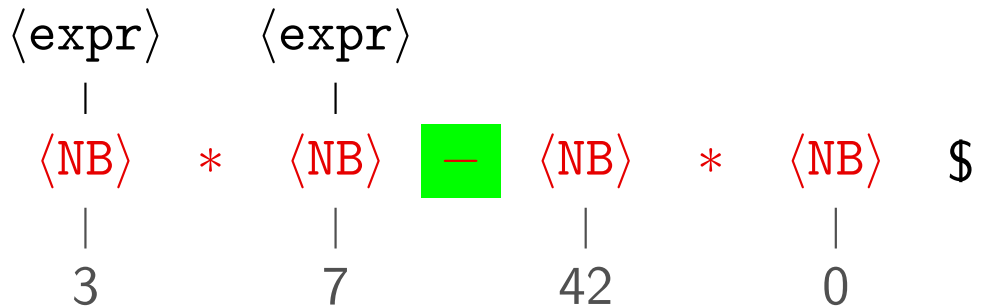
Automate du parseur LR en action



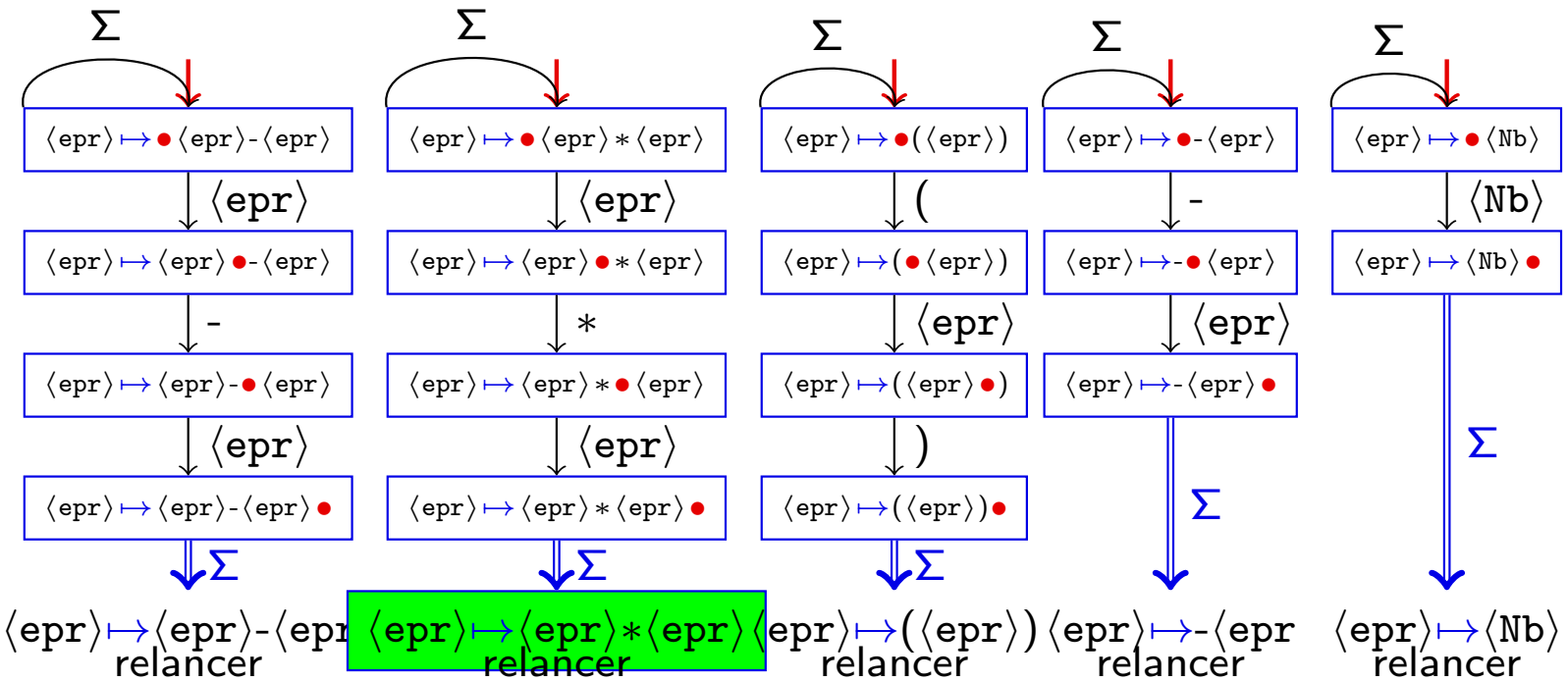


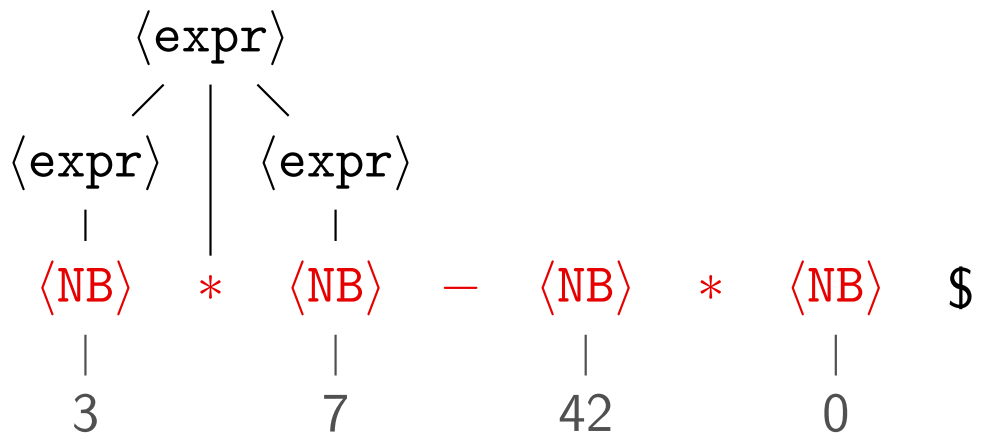
Automate du parseur LR en action



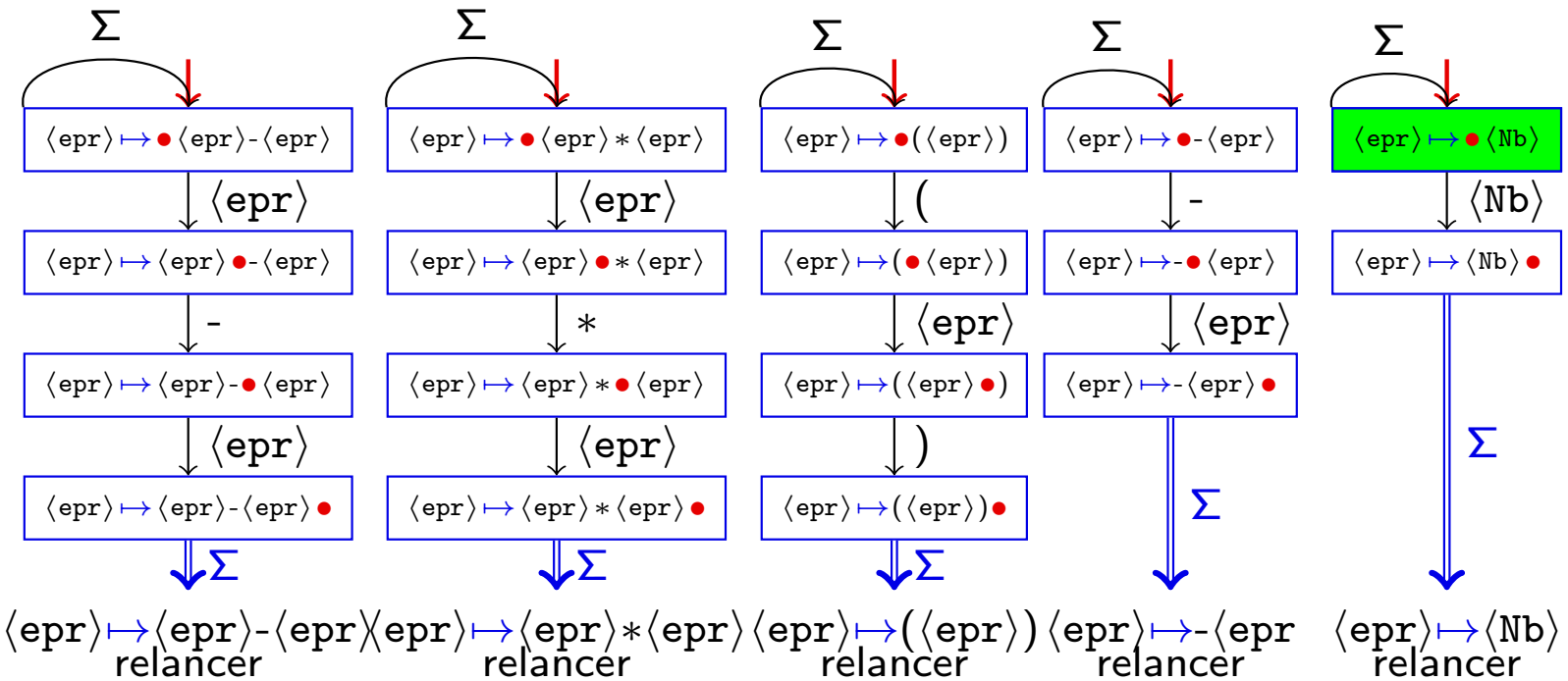


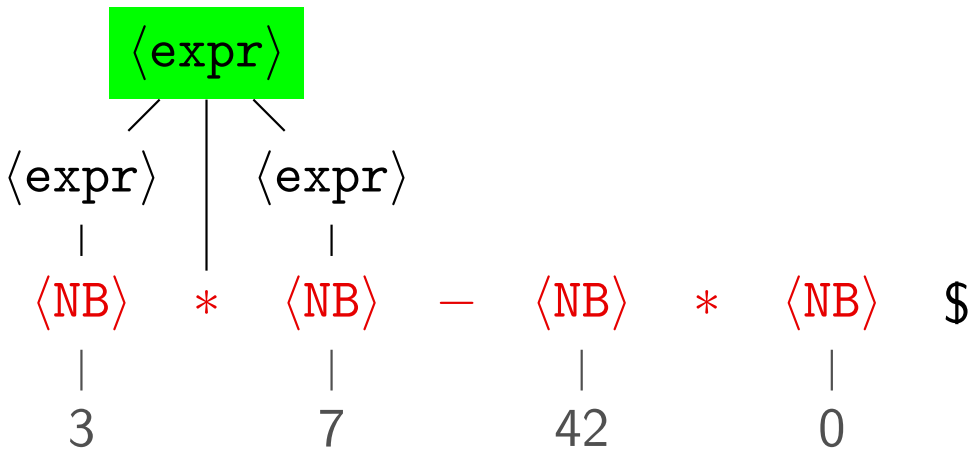
Automate du parseur LR en action



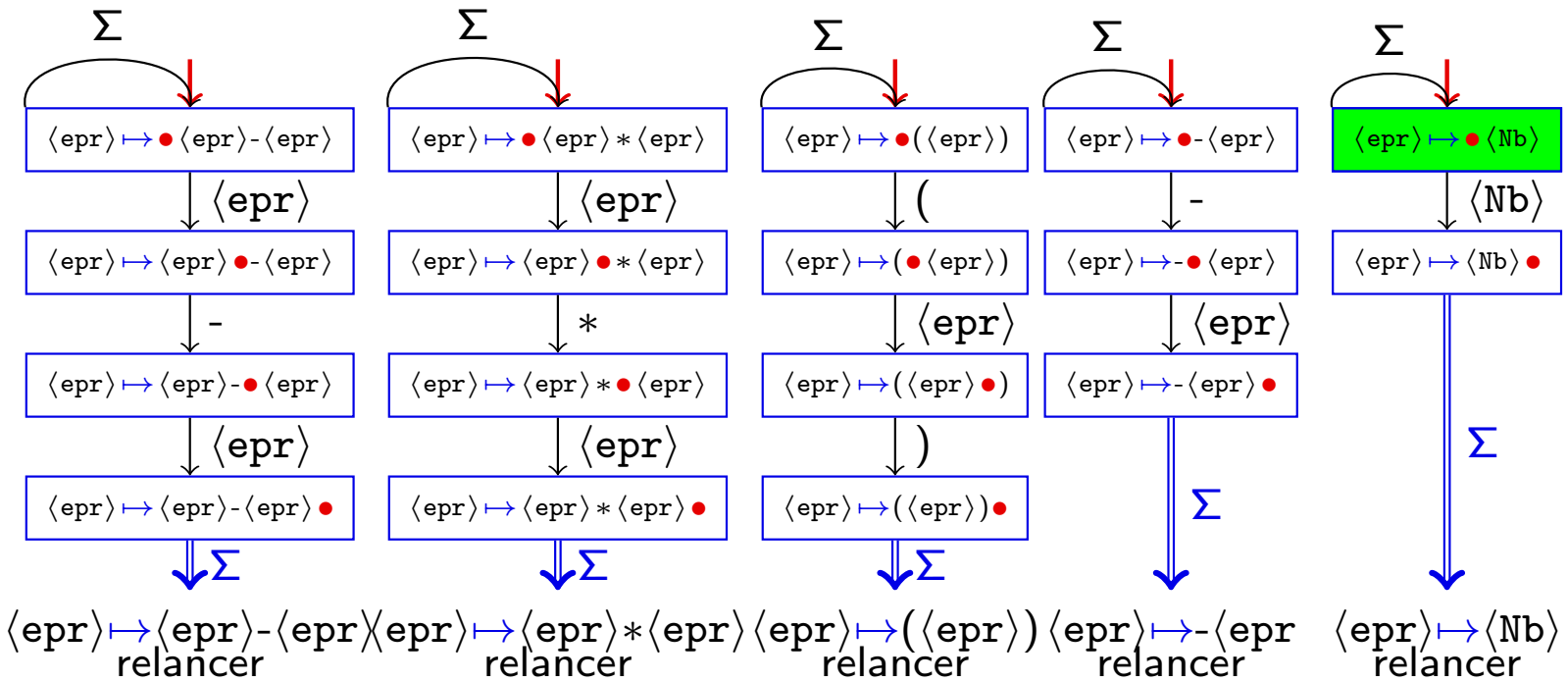


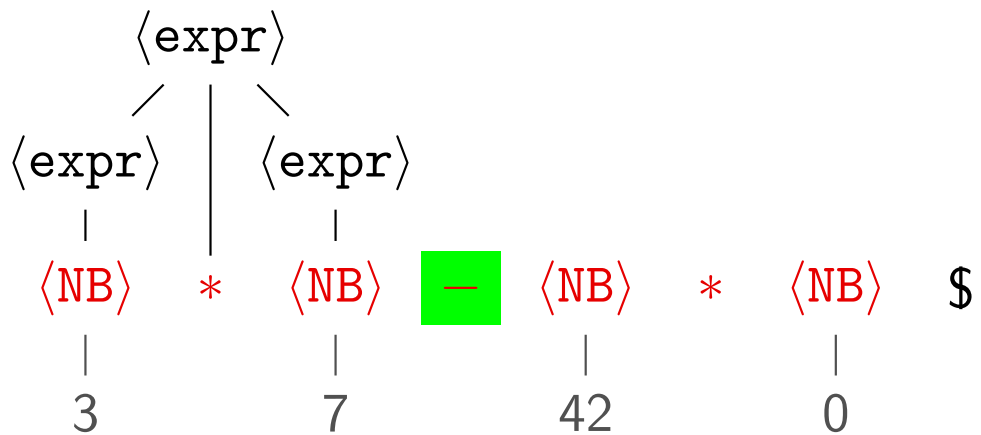
Automate du parseur LR en action



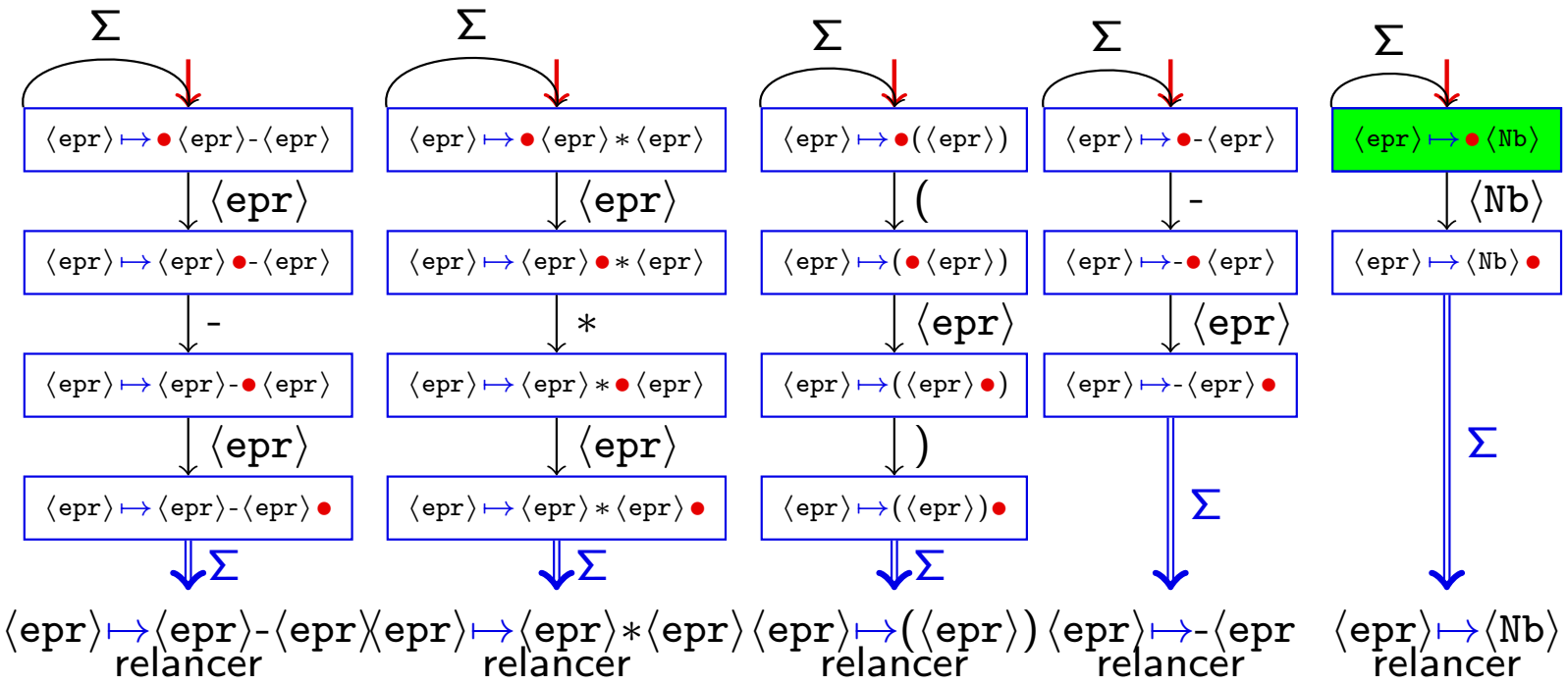


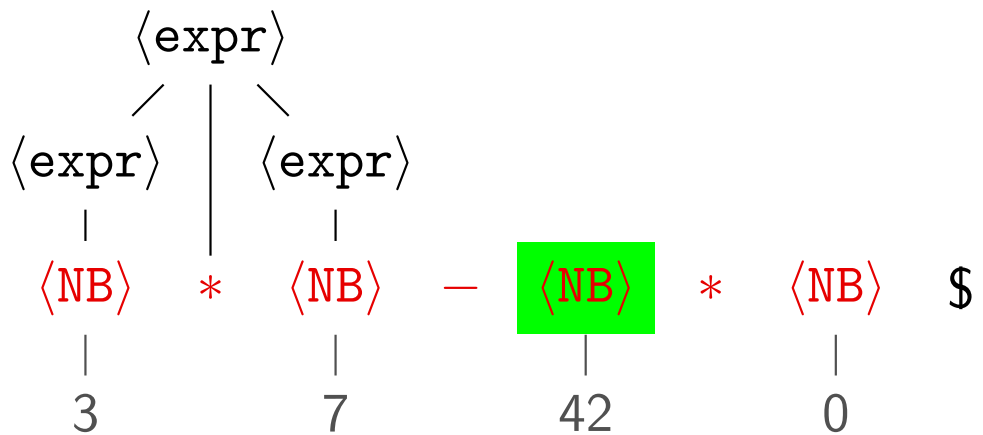
Automate du parseur LR en action



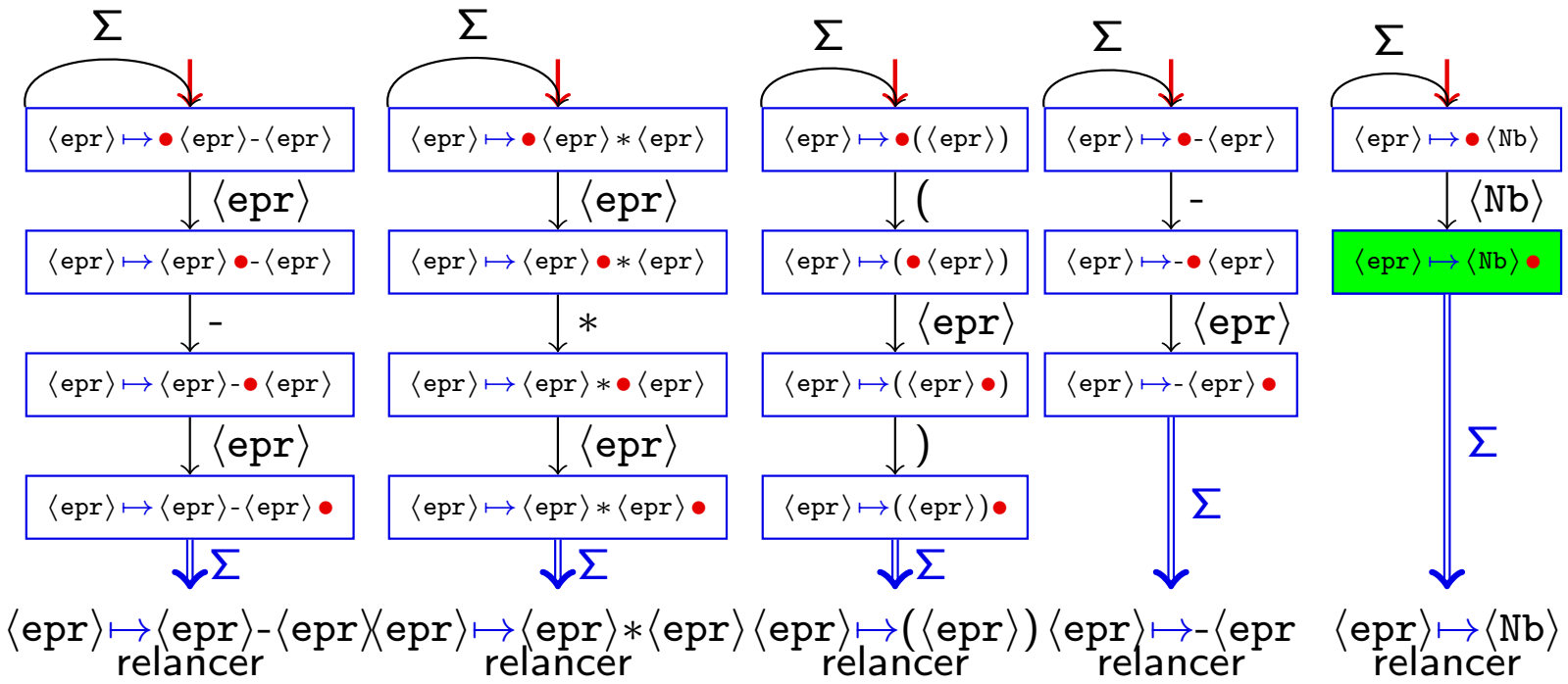


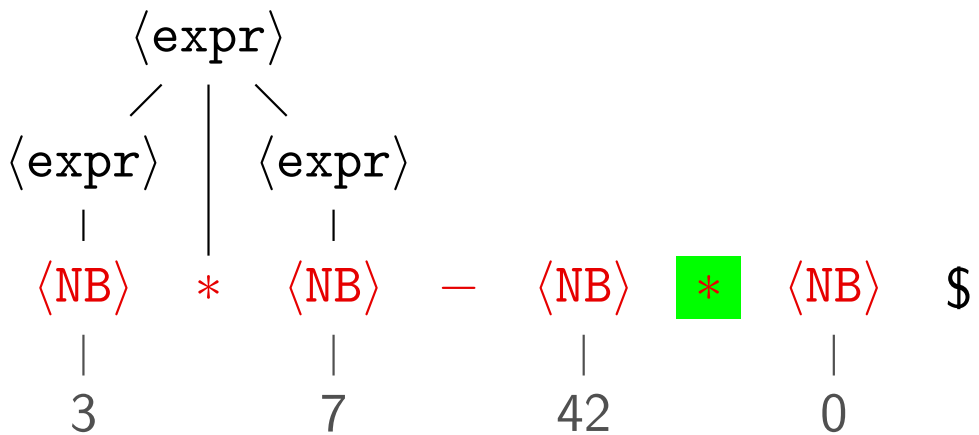
Automate du parseur LR en action



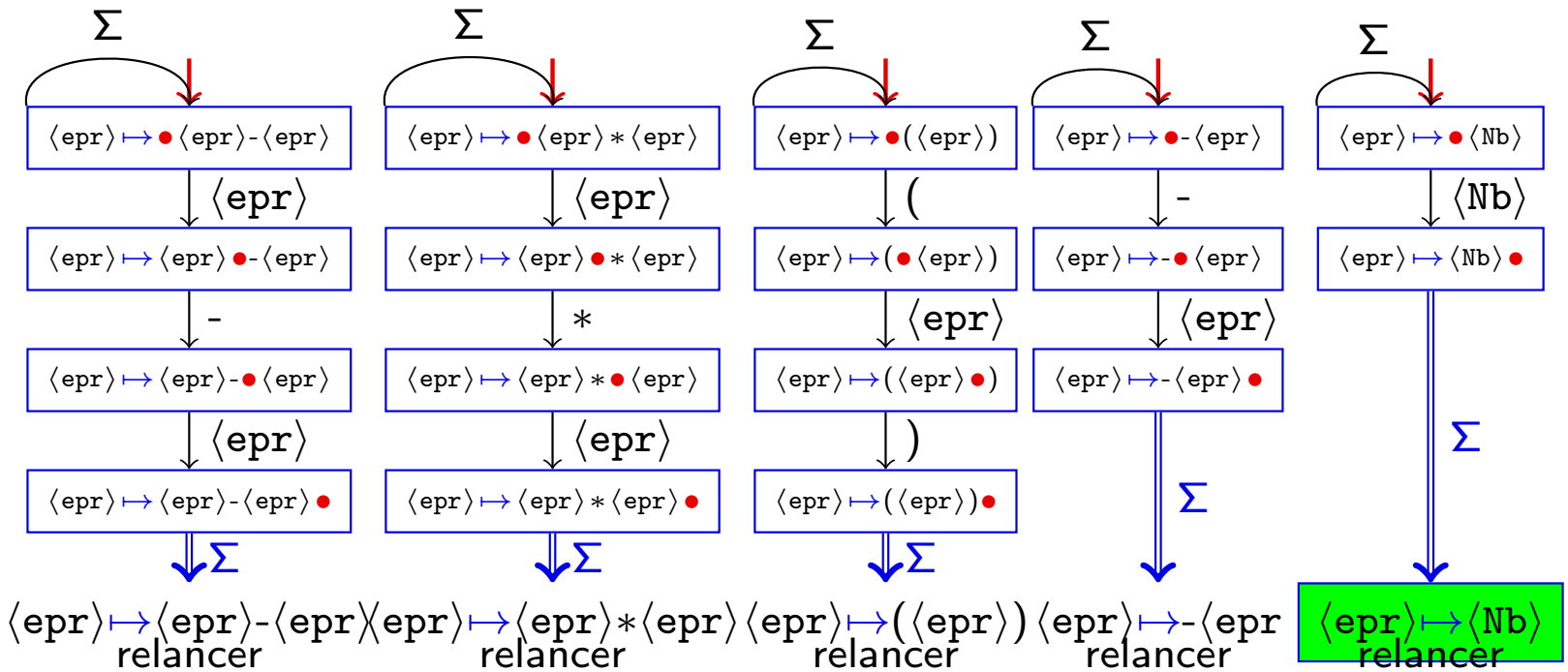


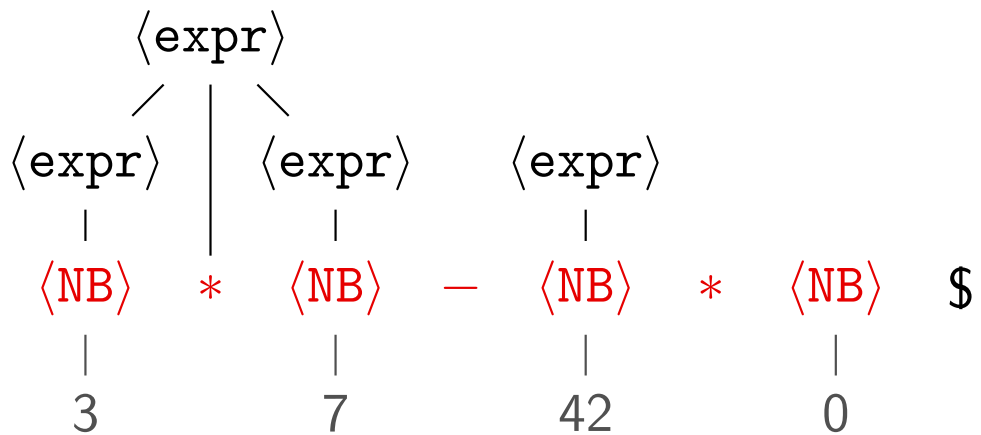
Automate du parseur LR en action



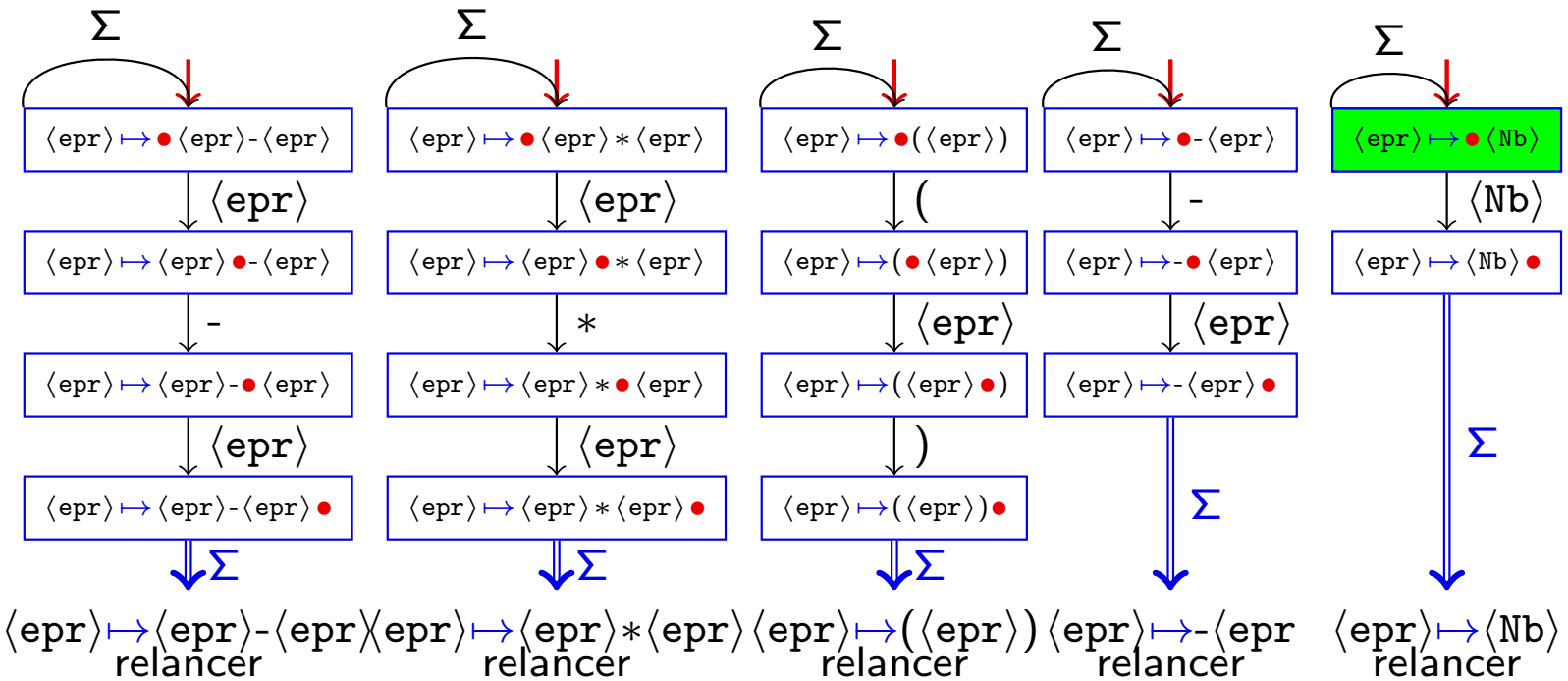


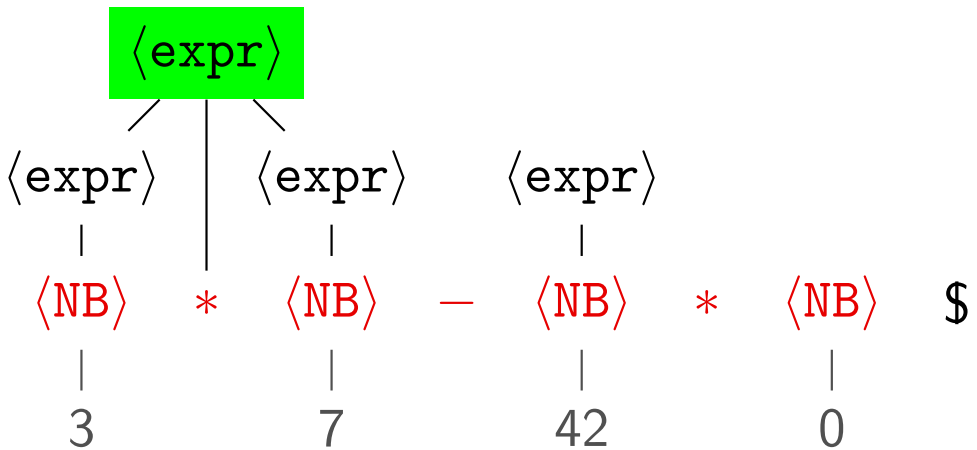
Automate du parseur LR en action



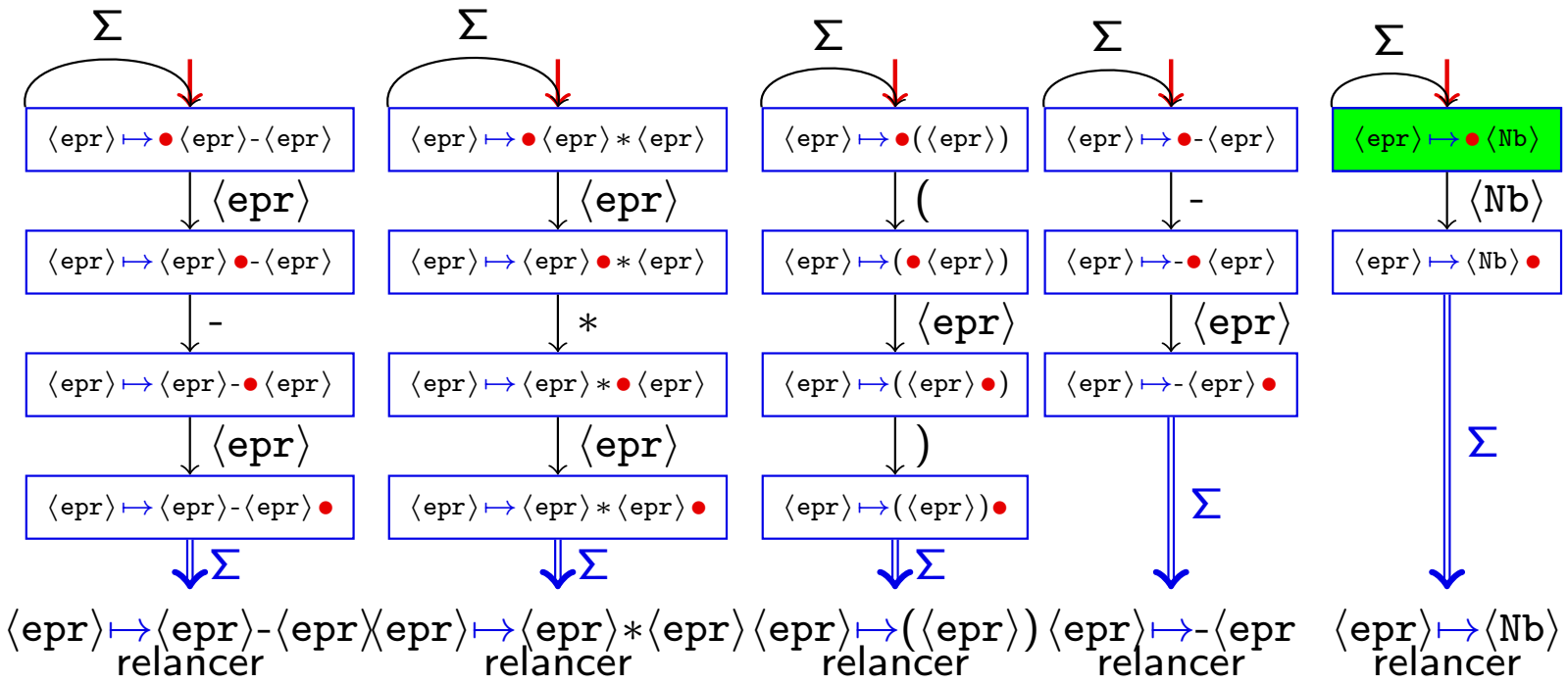


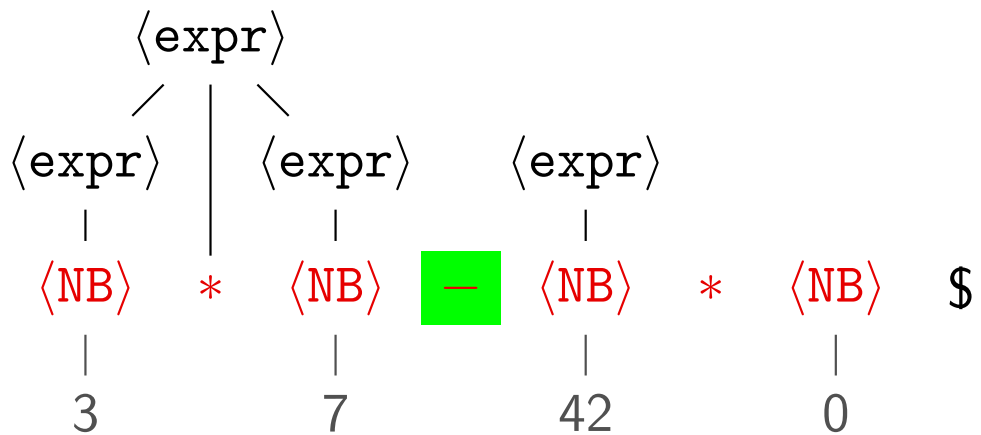
Automate du parseur LR en action



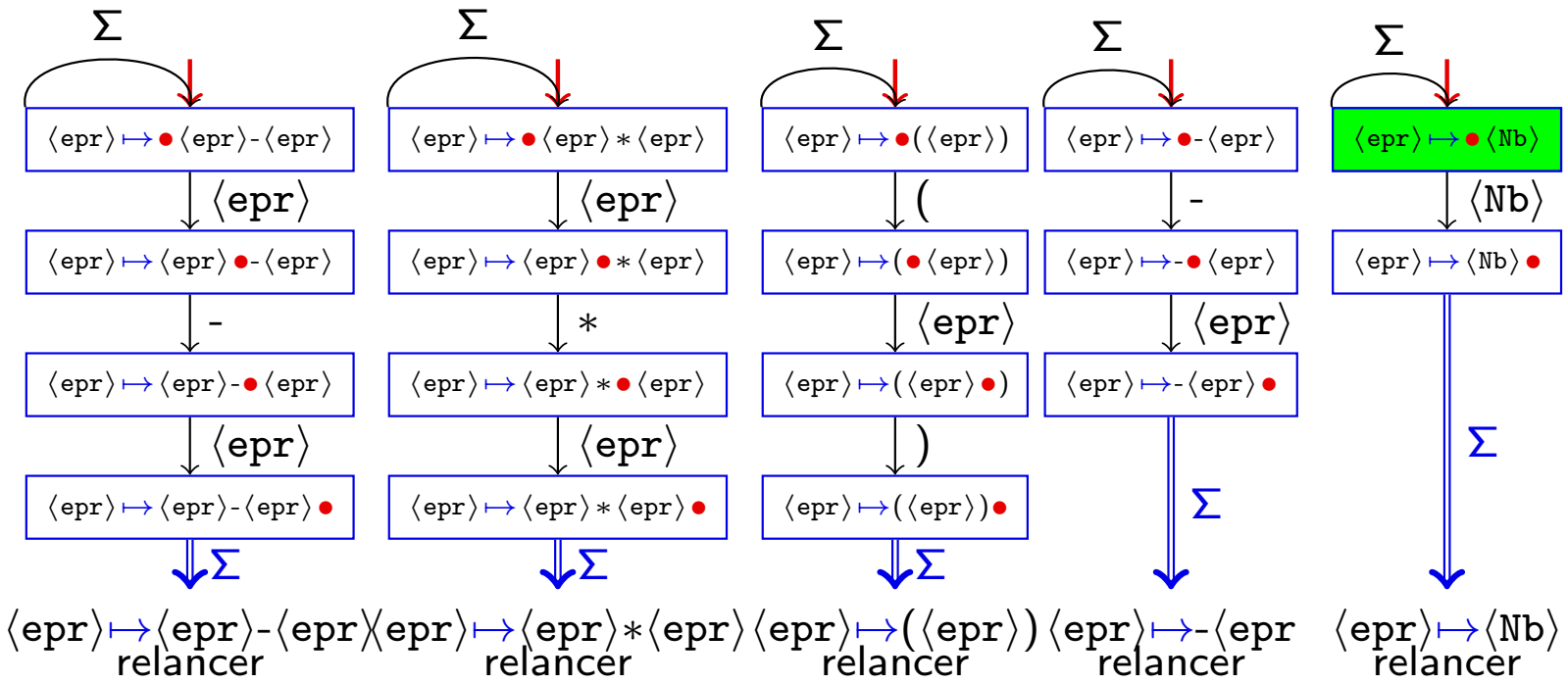


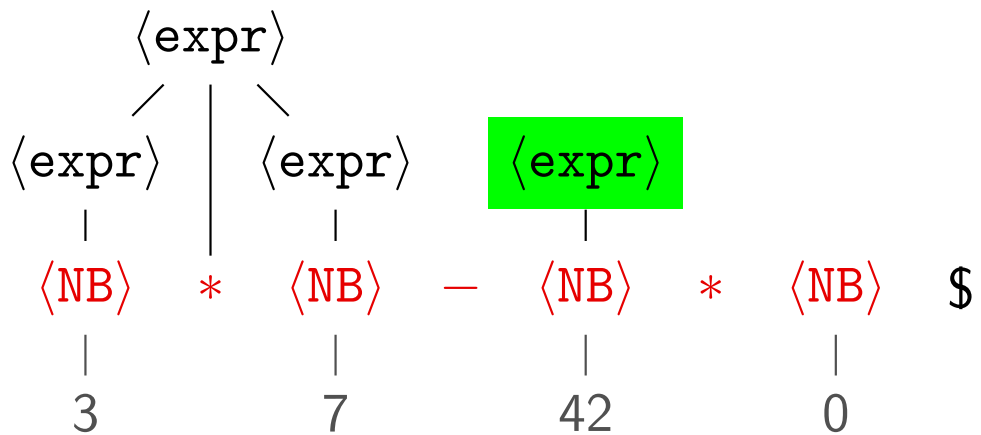
Automate du parseur LR en action



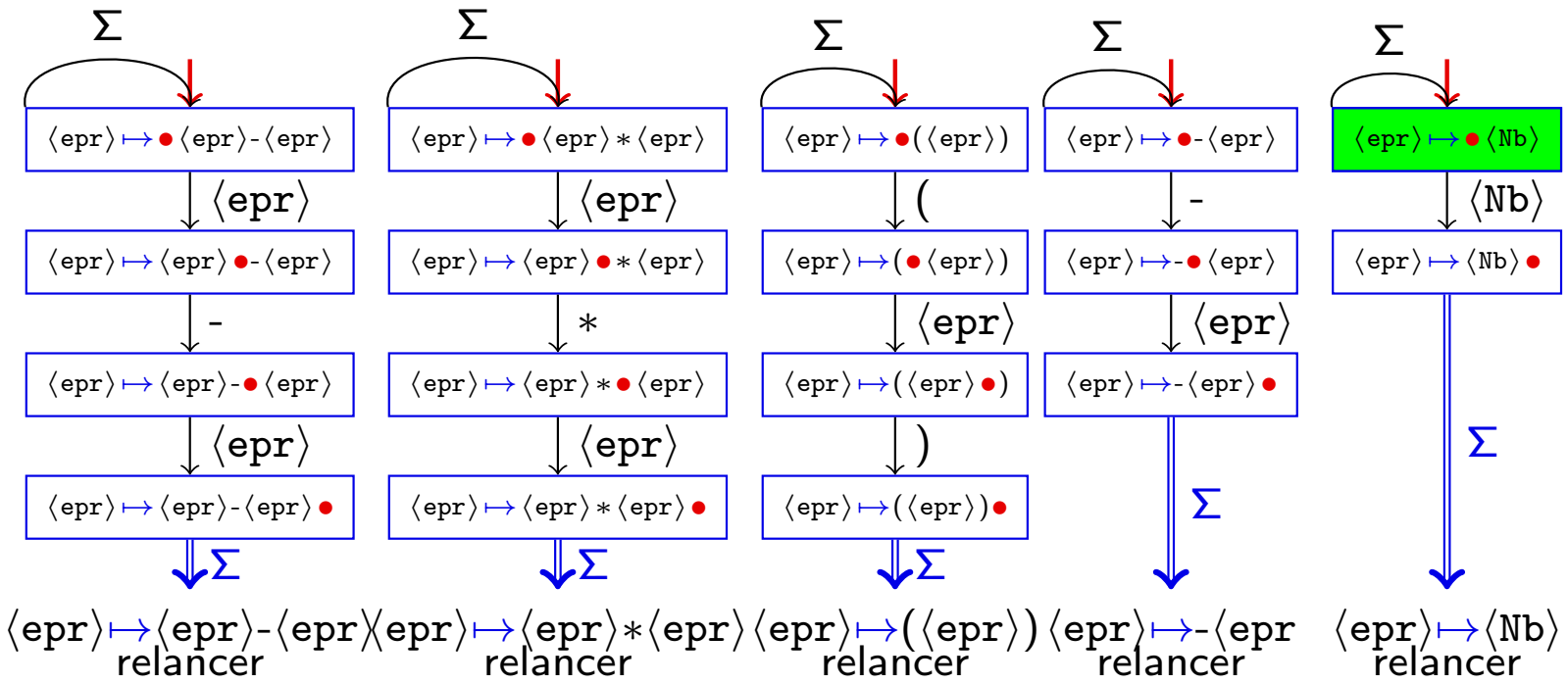


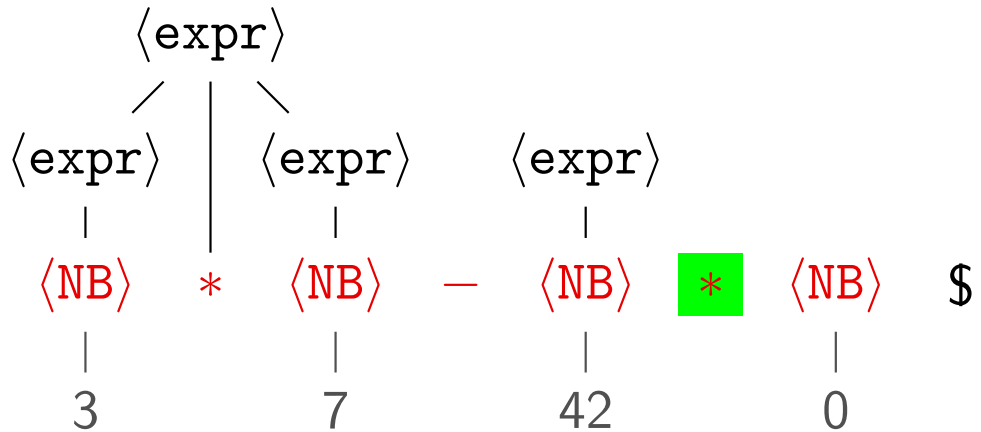
Automate du parseur LR en action



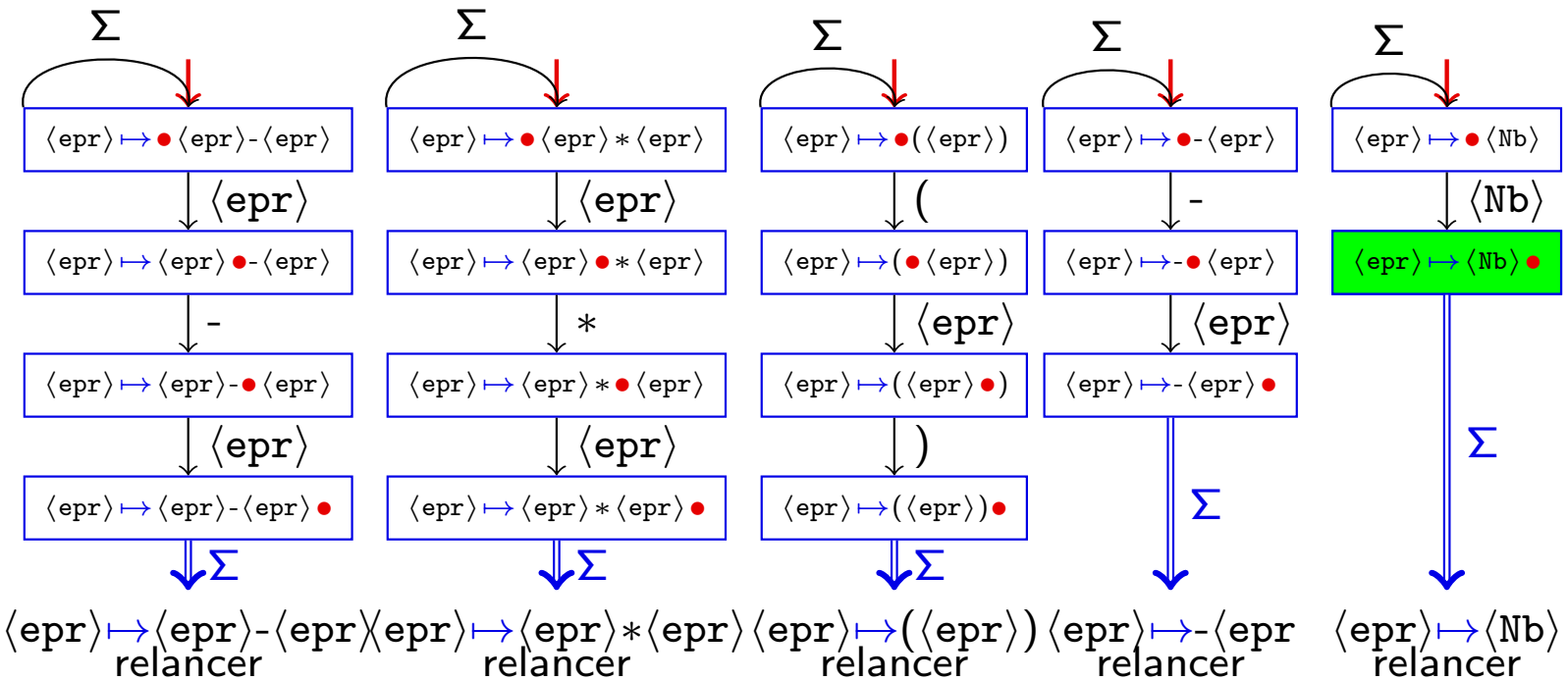


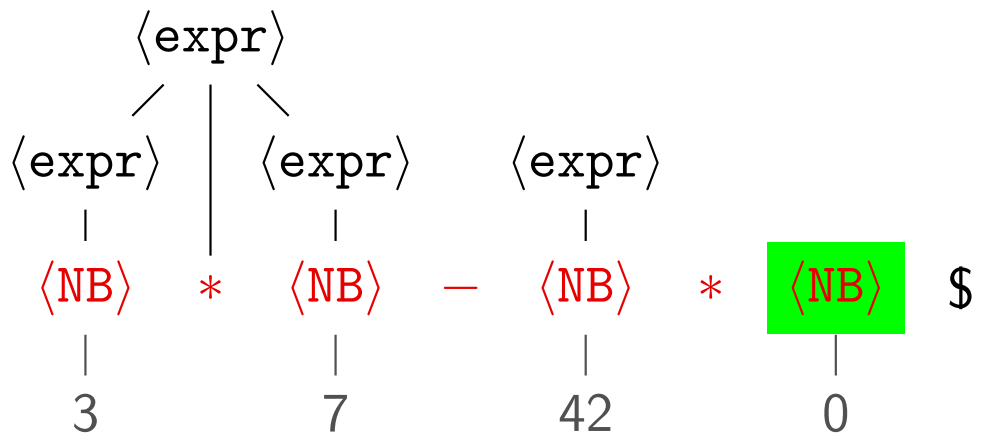
Automate du parseur LR en action



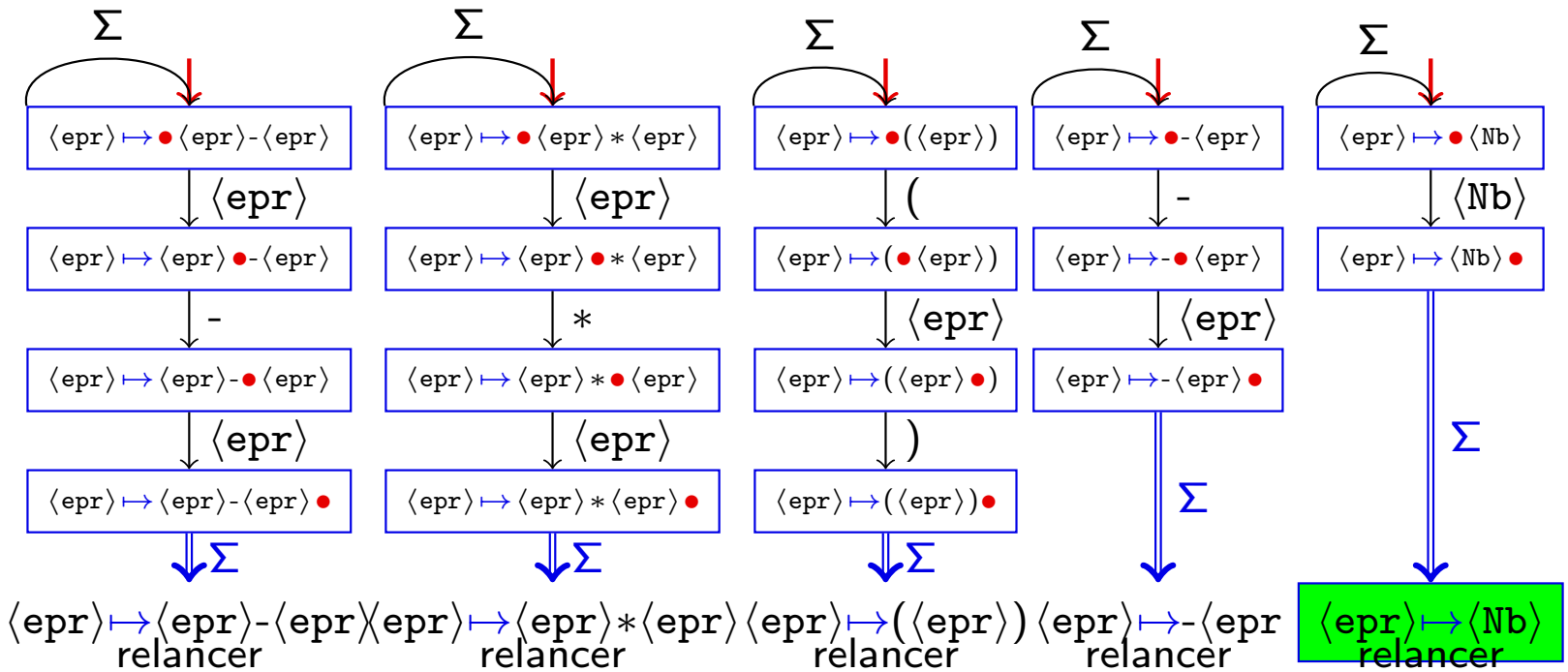


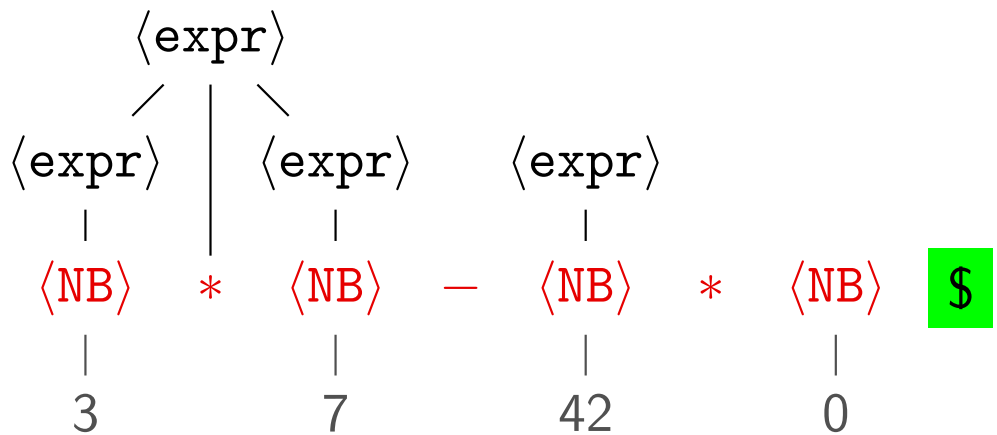
Automate du parseur LR en action



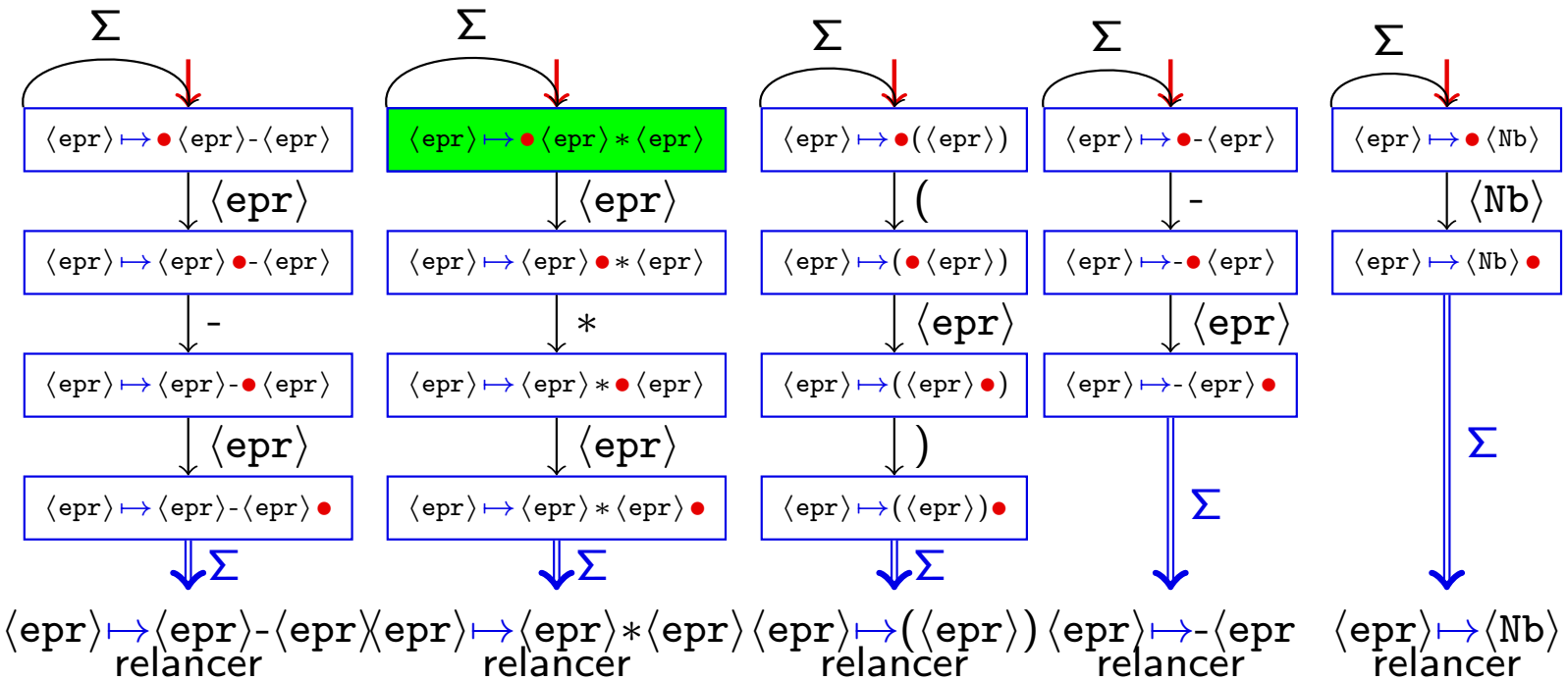


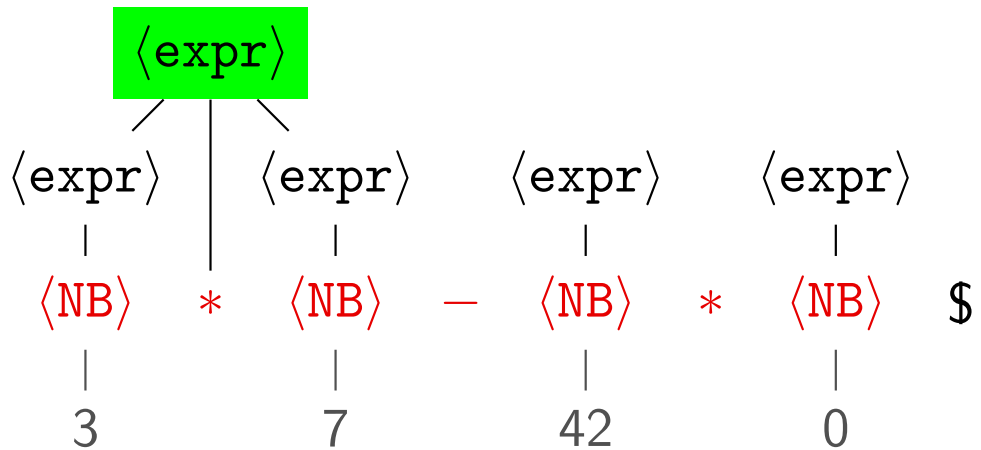
Automate du parseur LR en action



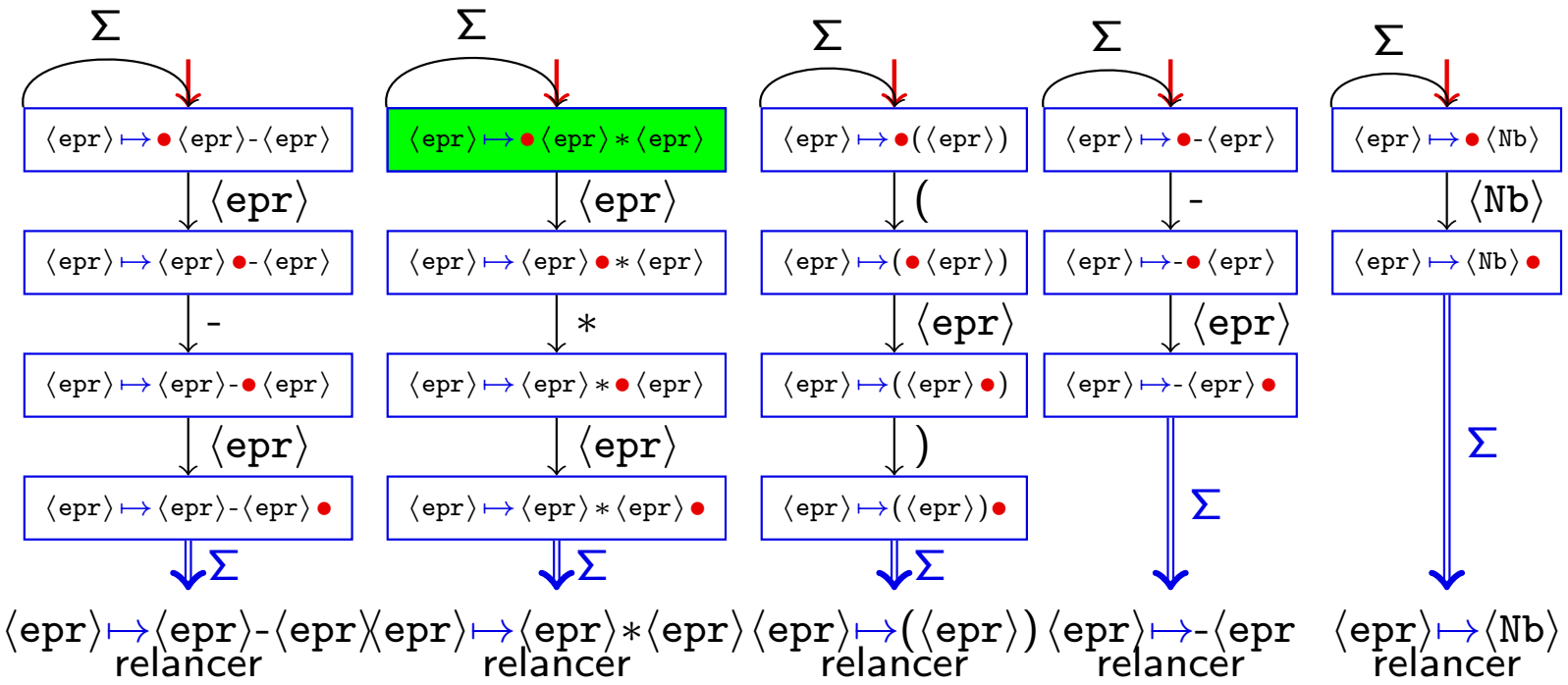


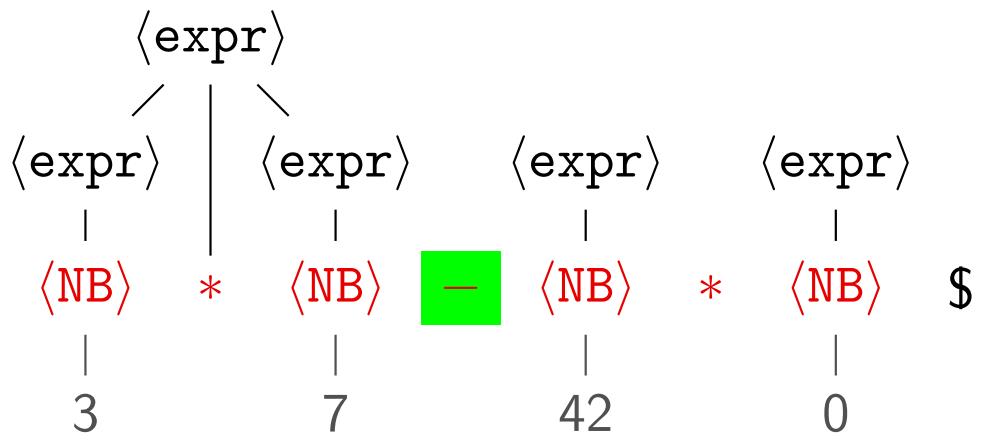
Automate du parseur LR en action



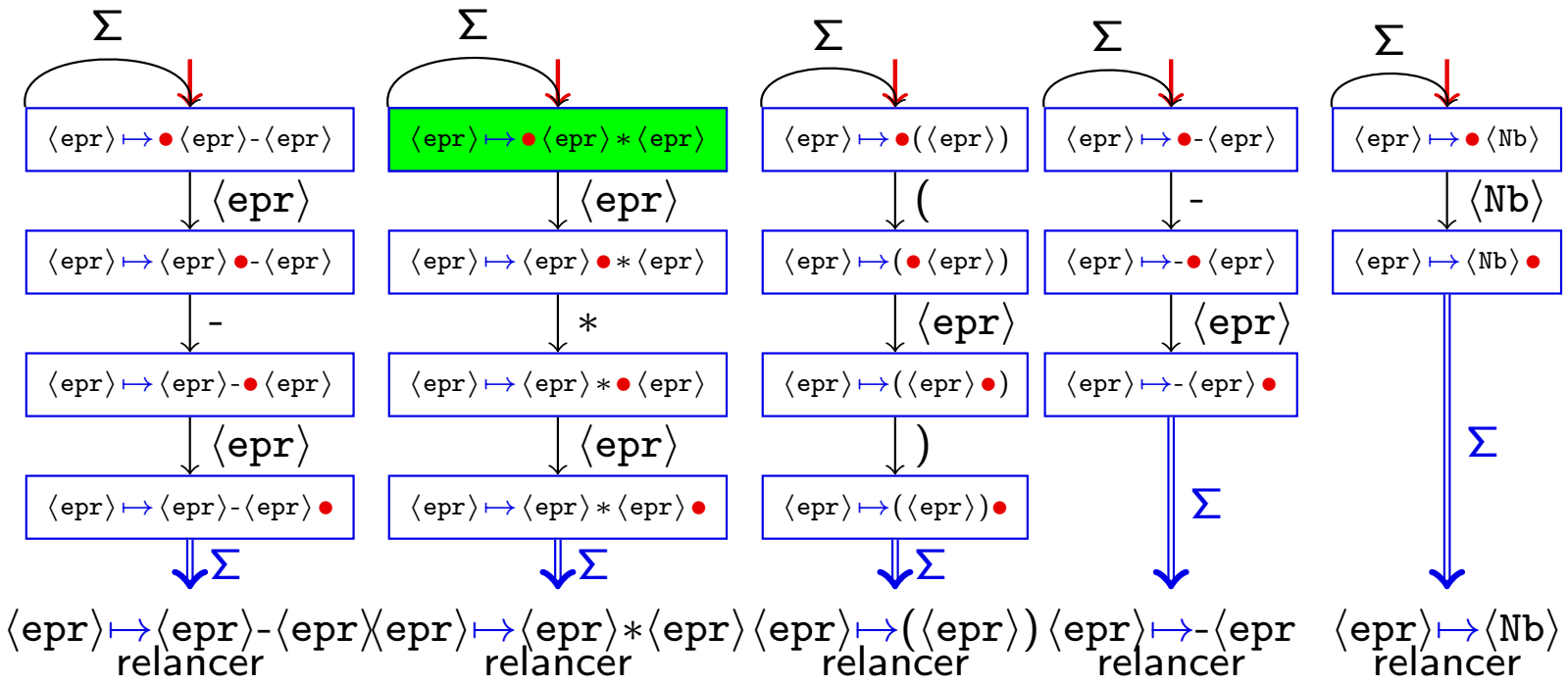


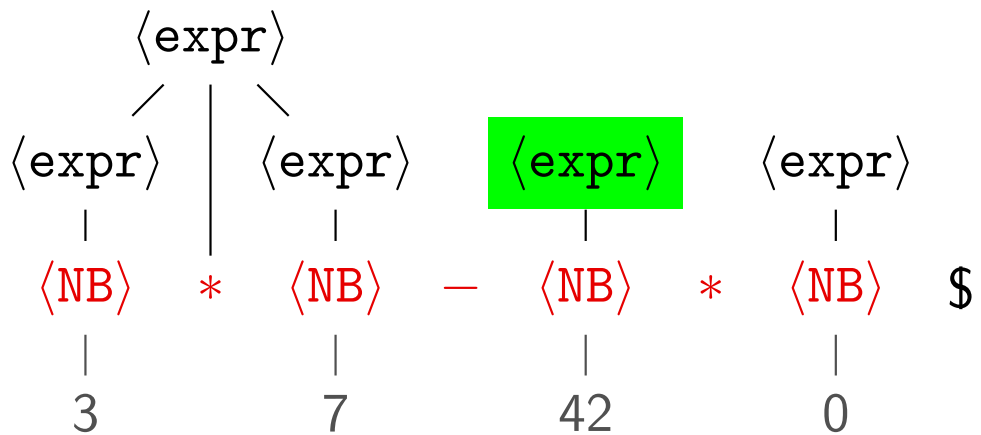
Automate du parseur LR en action



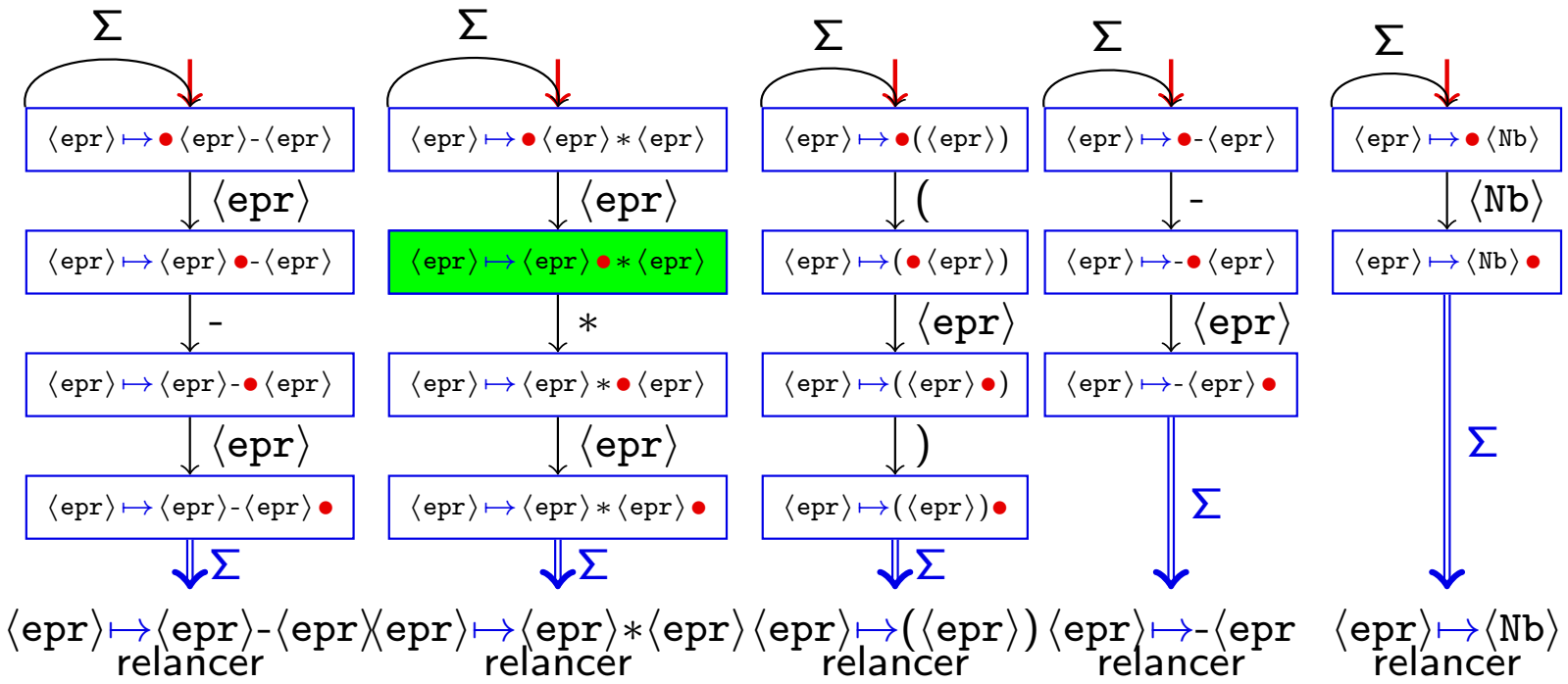


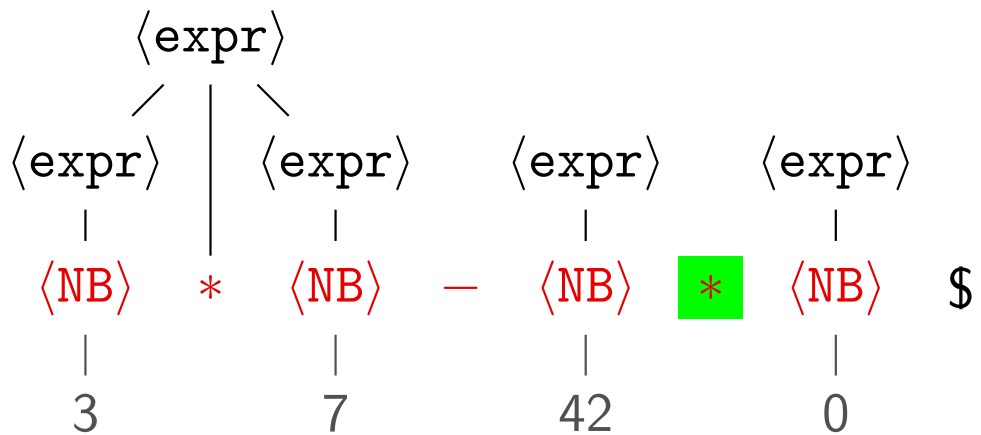
Automate du parseur LR en action



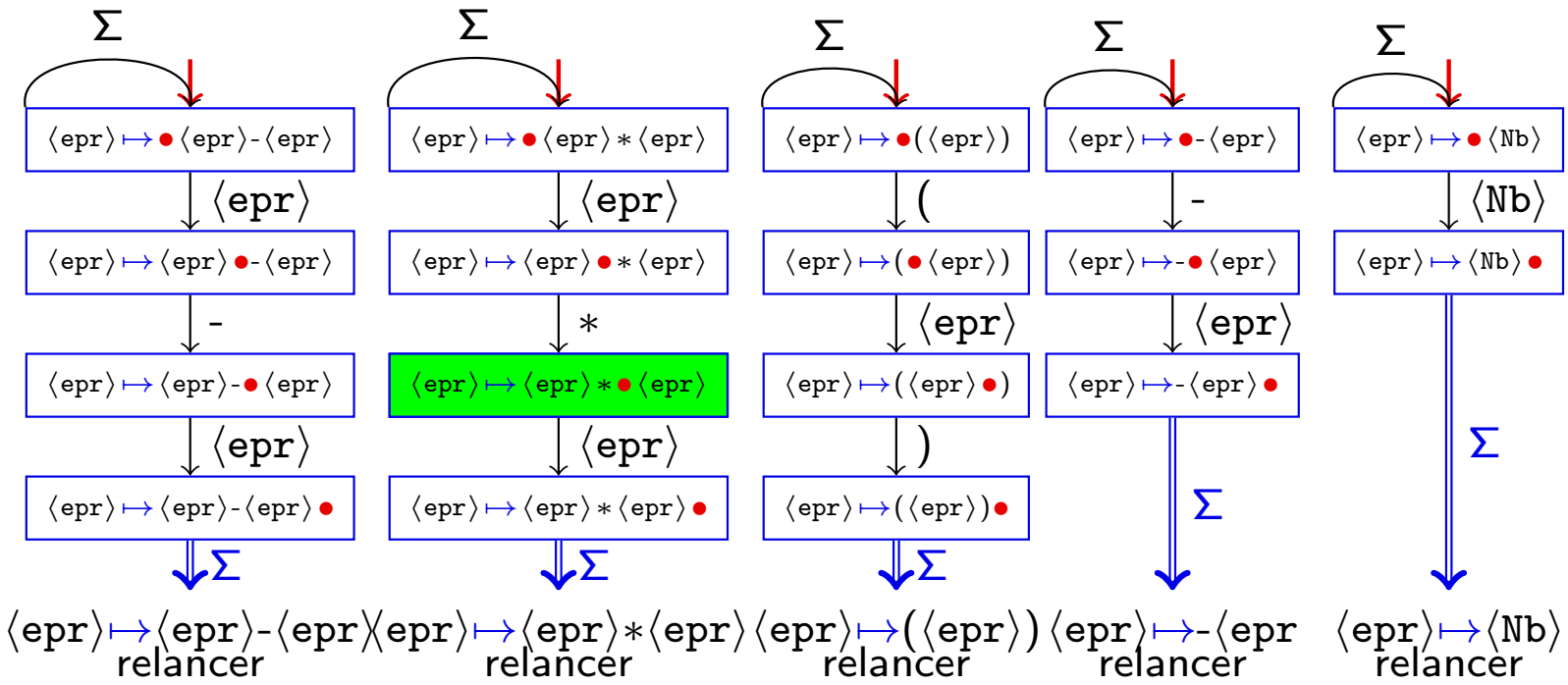


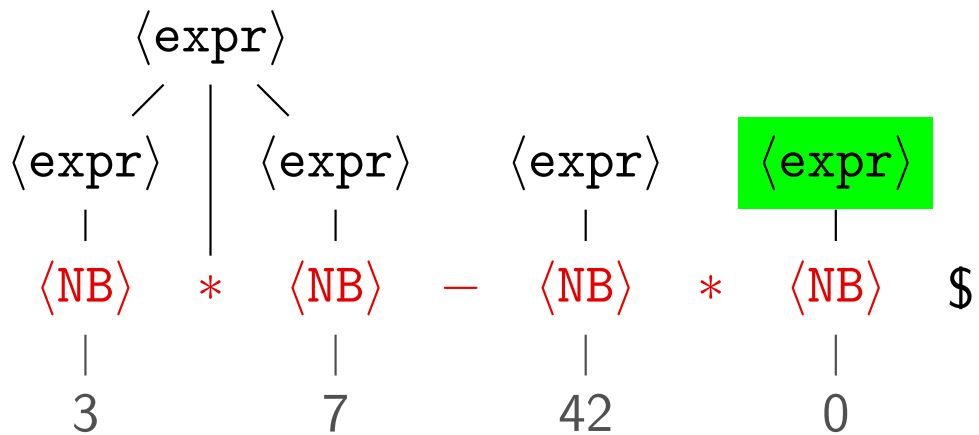
Automate du parseur LR en action



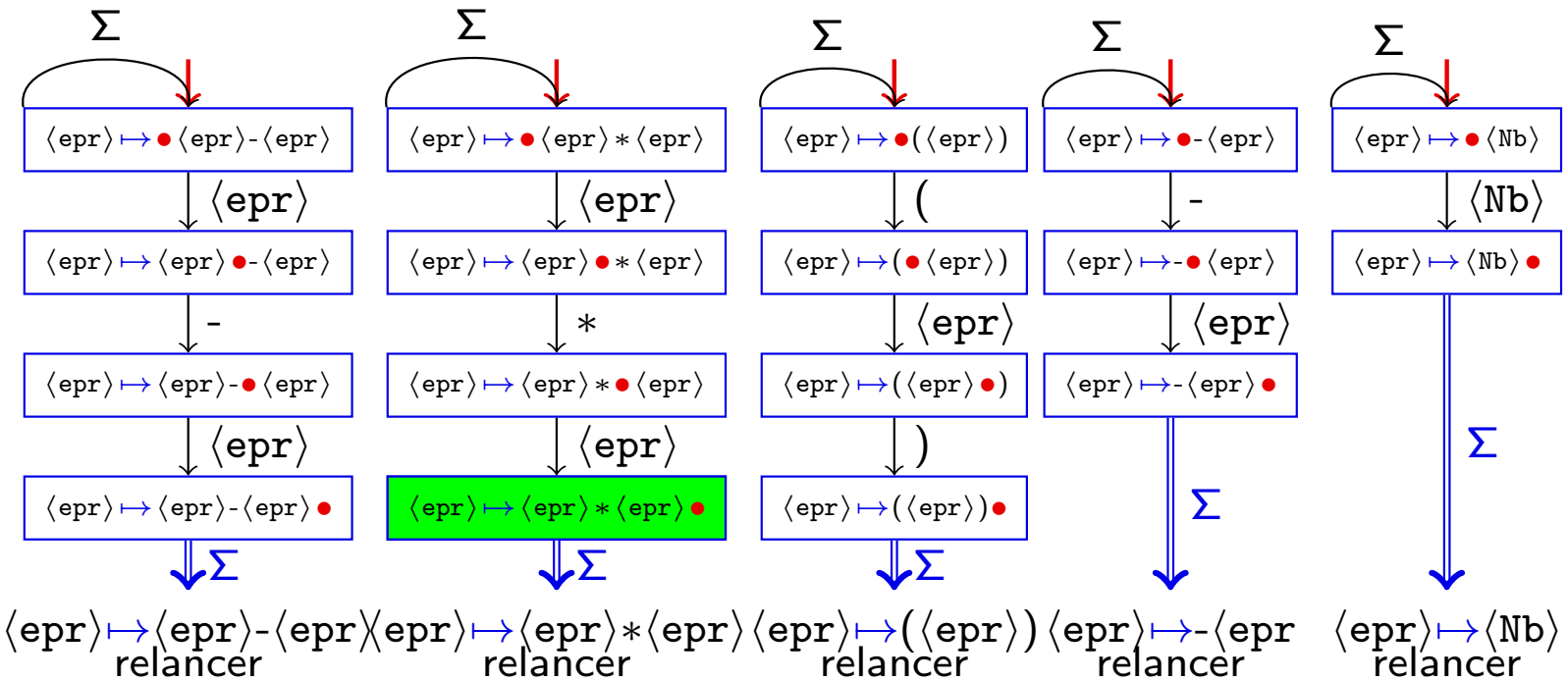


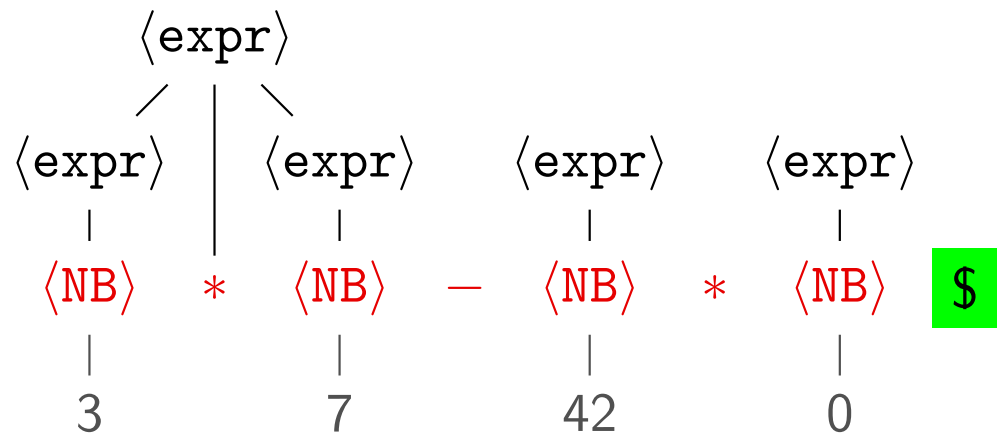
Automate du parseur LR en action



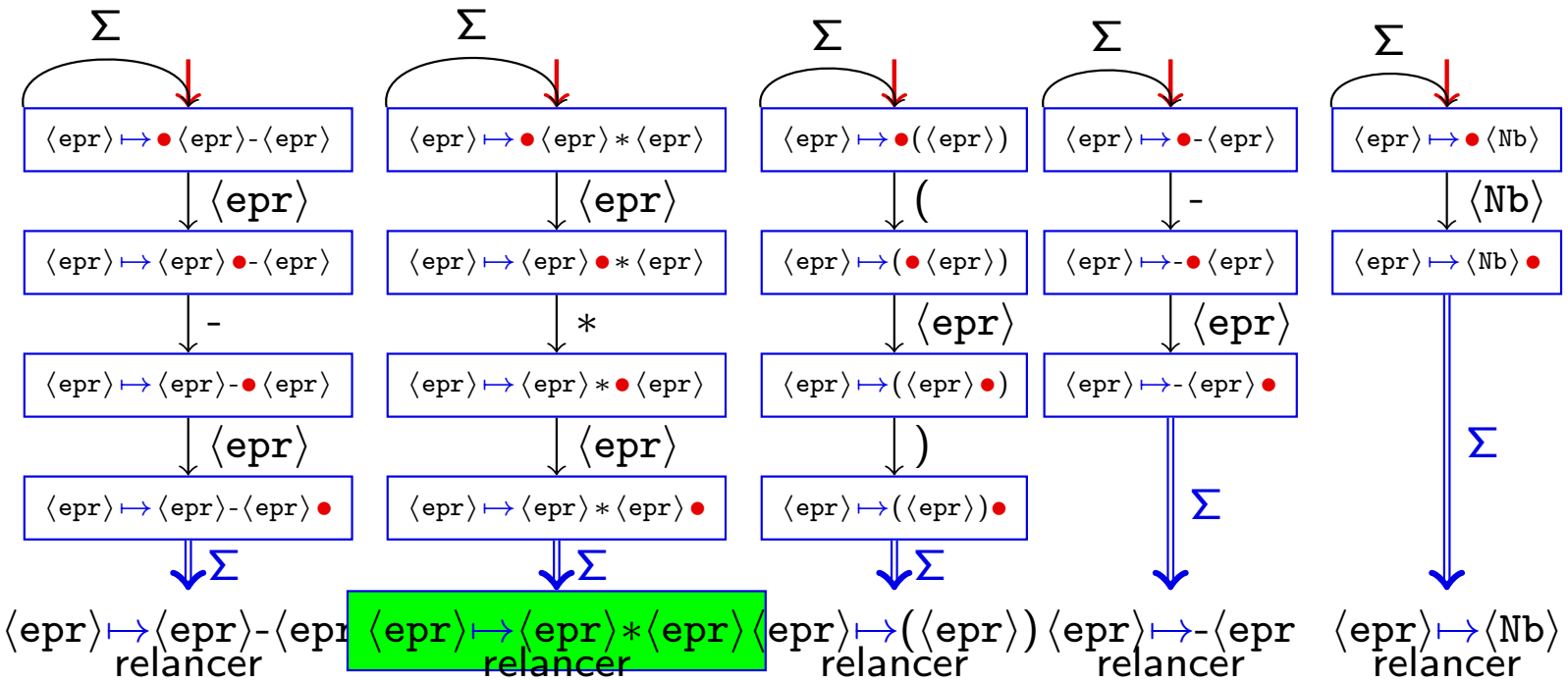


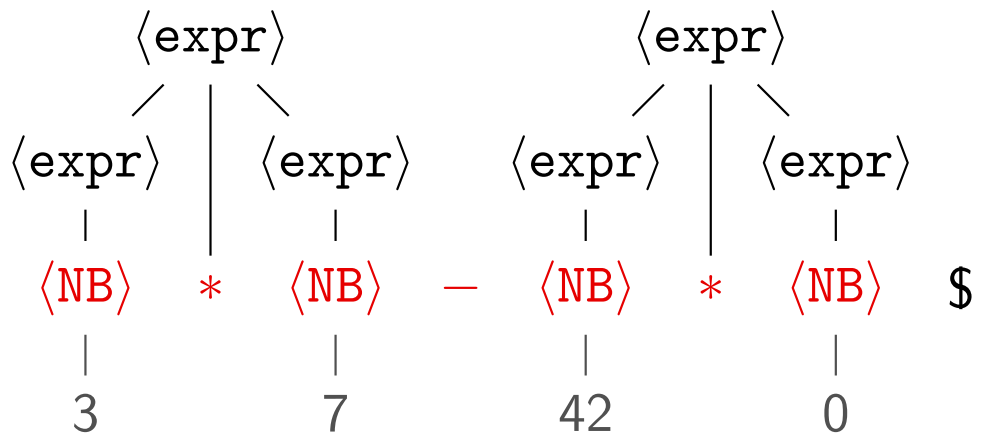
Automate du parseur LR en action



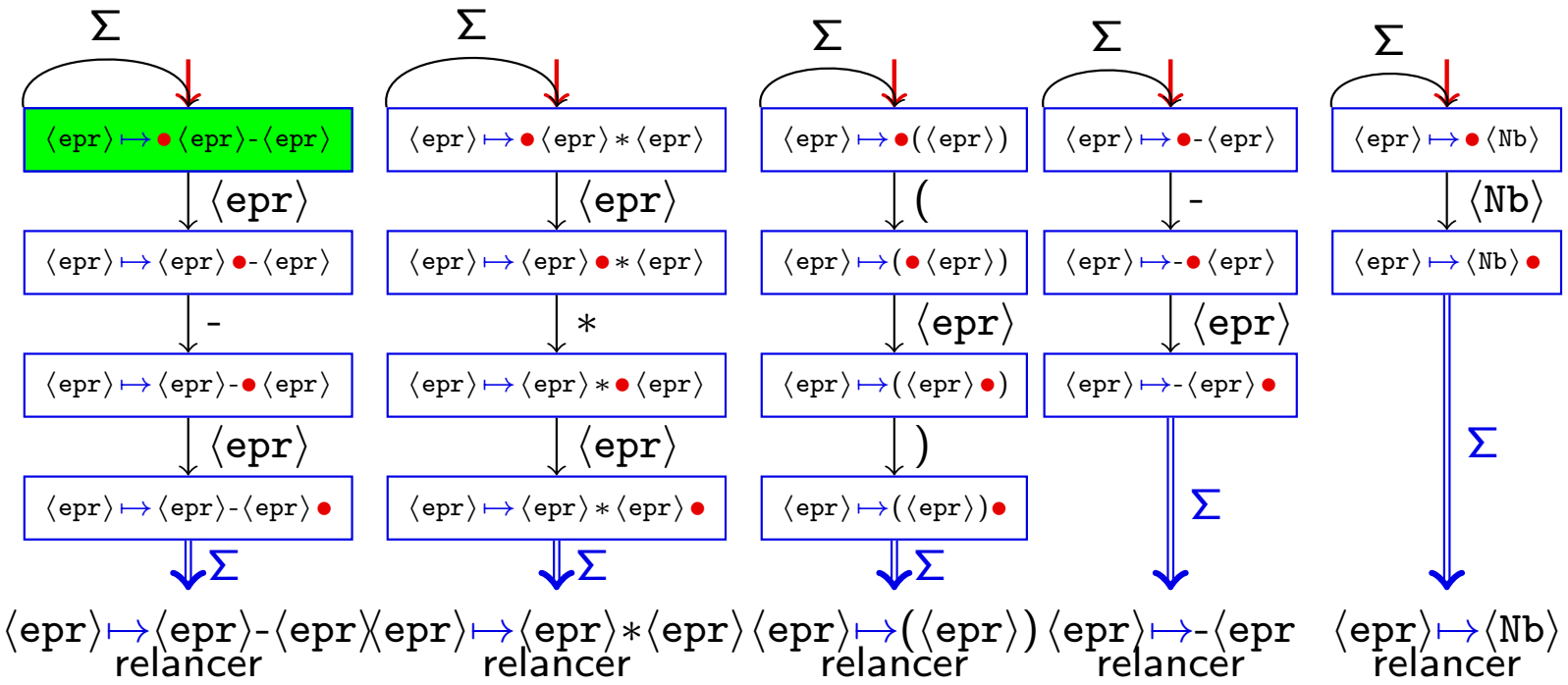


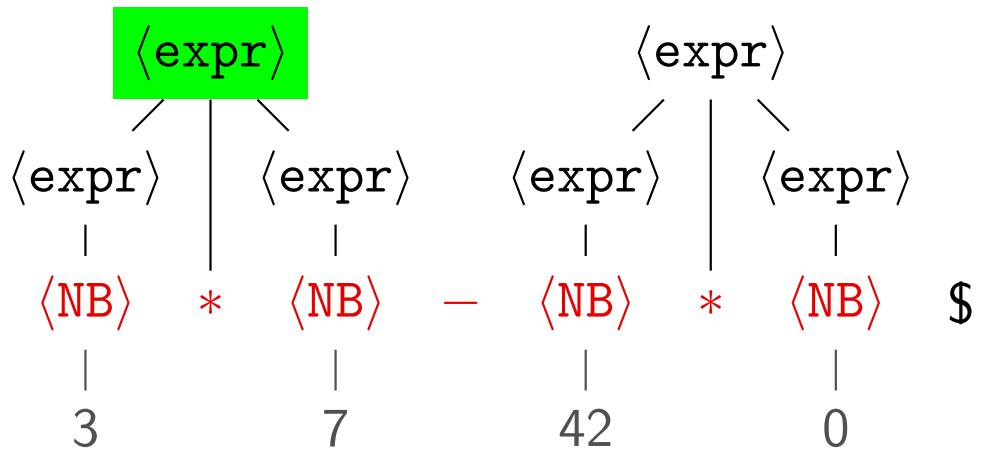
Automate du parseur LR en action



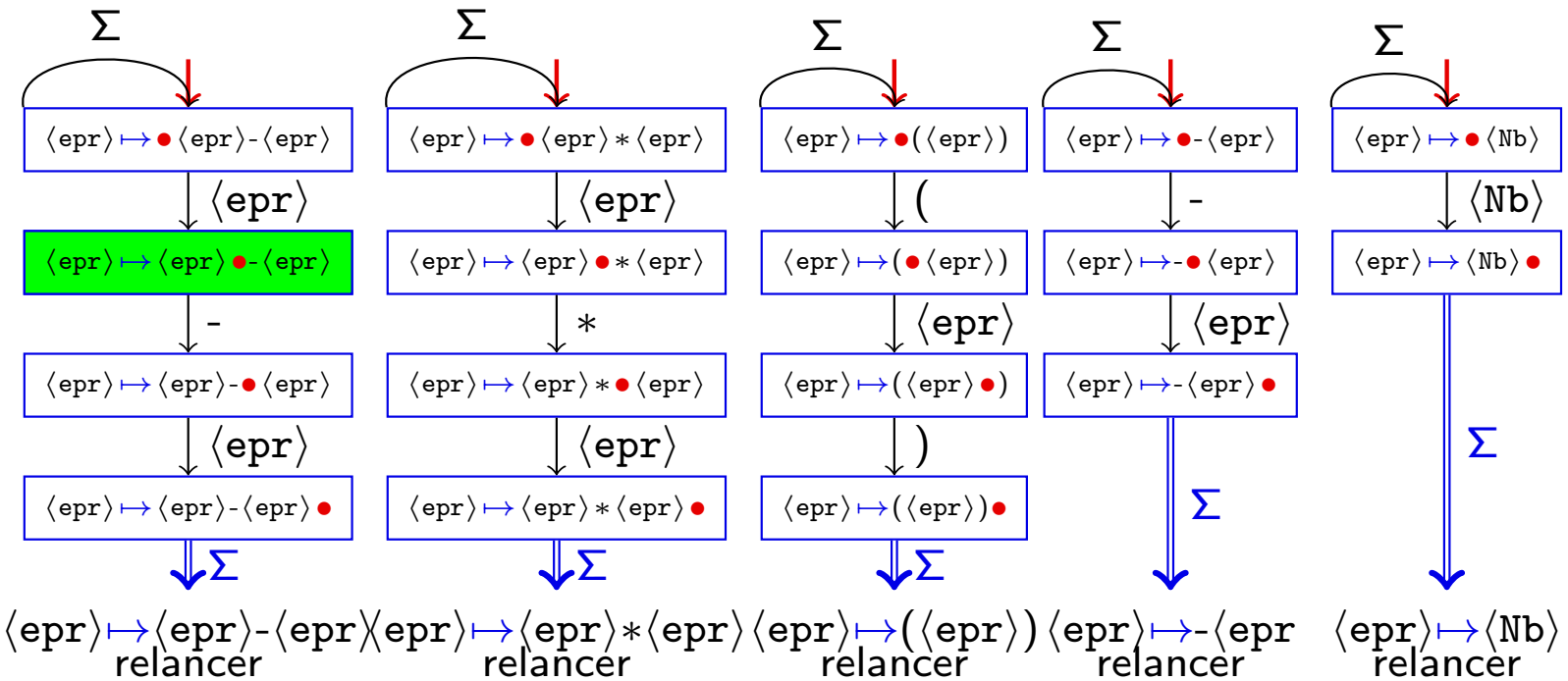


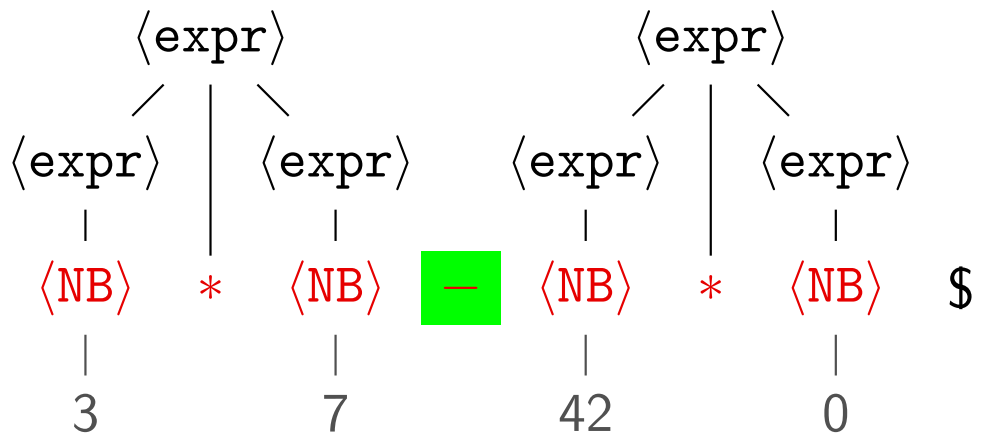
Automate du parseur LR en action



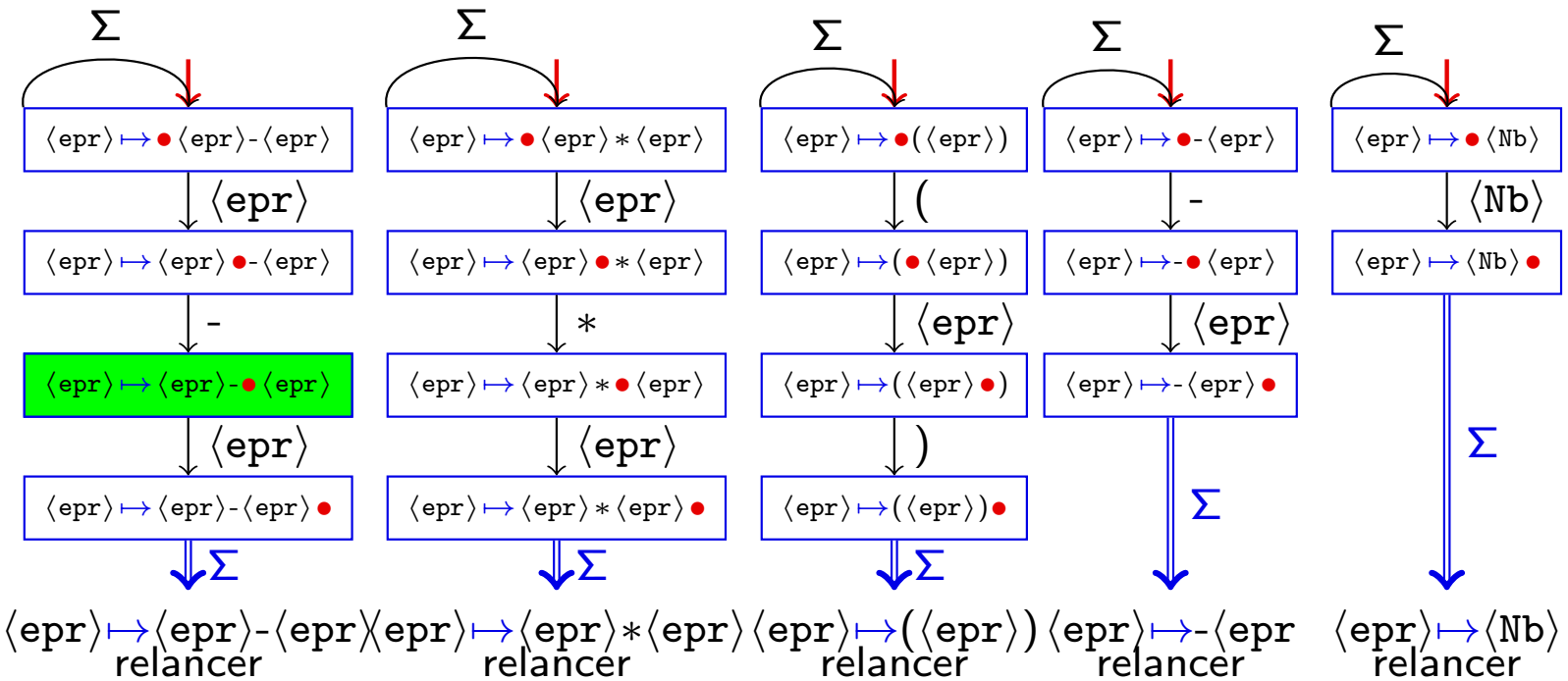


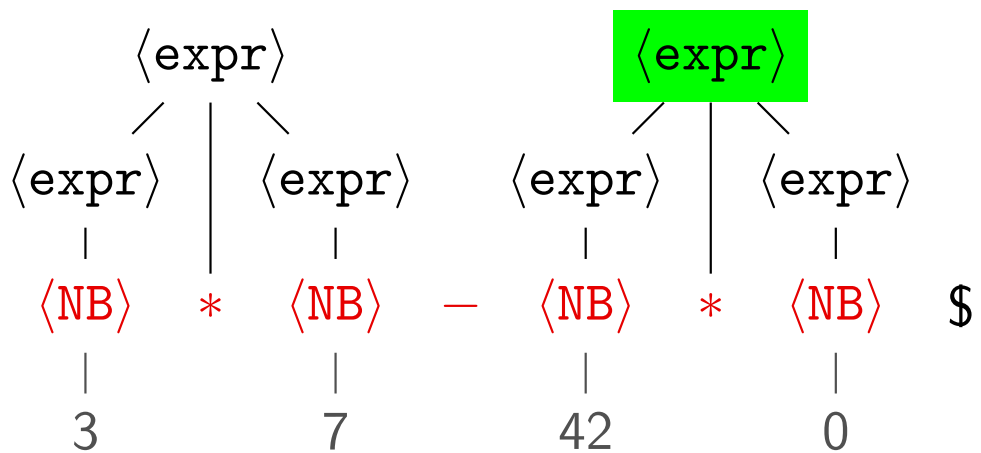
Automate du parseur LR en action



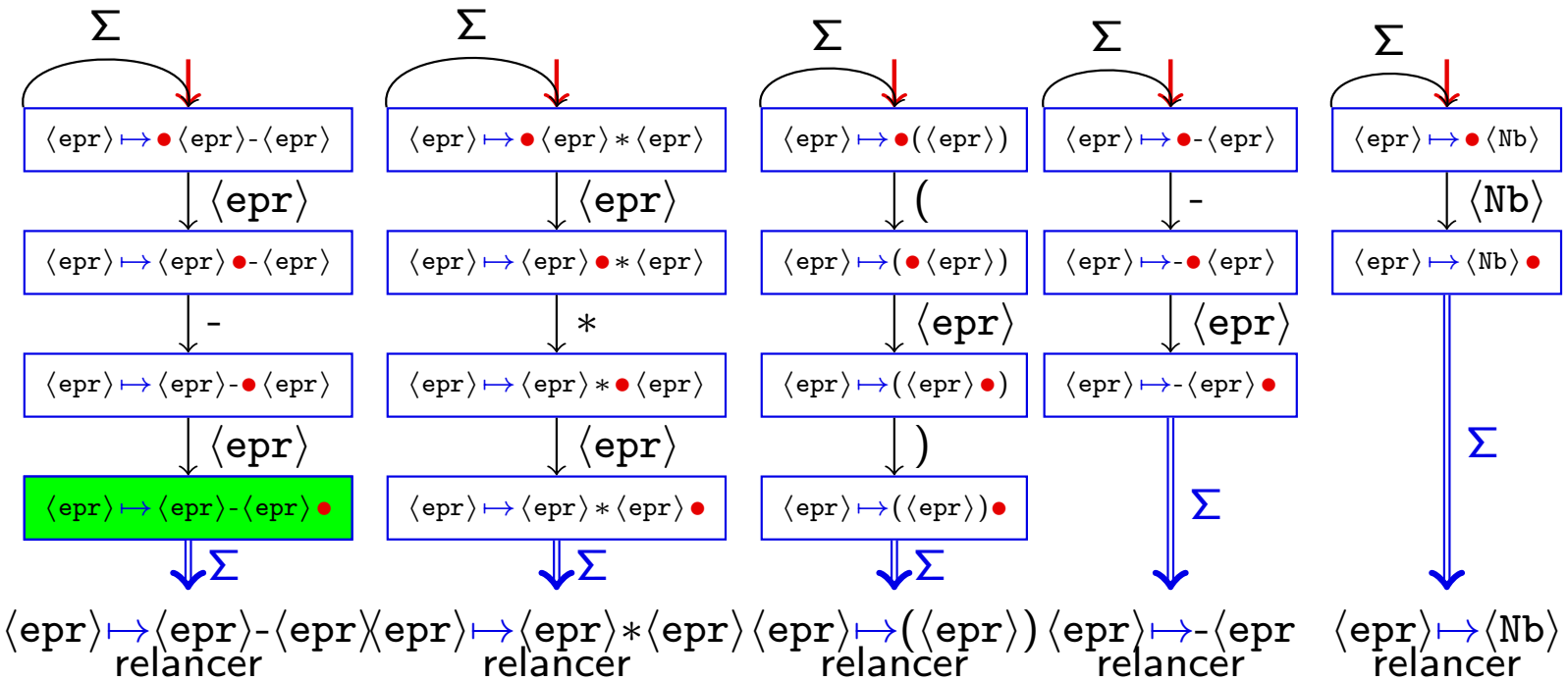


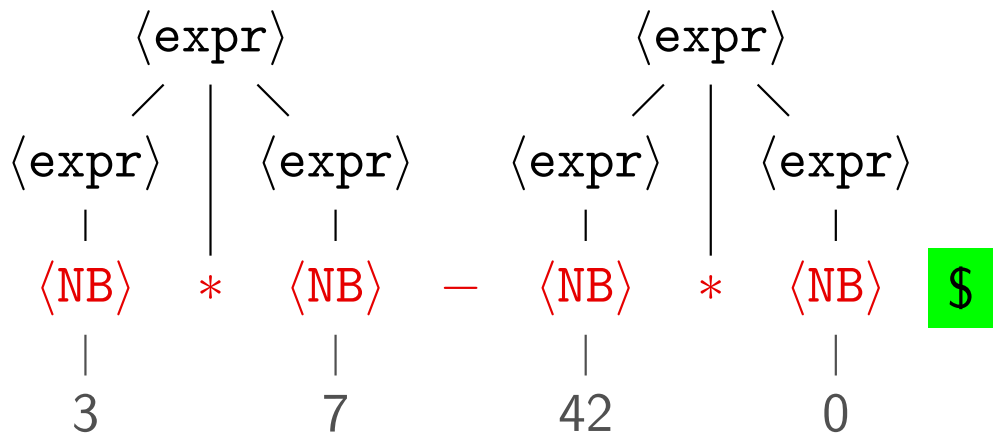
Automate du parseur LR en action



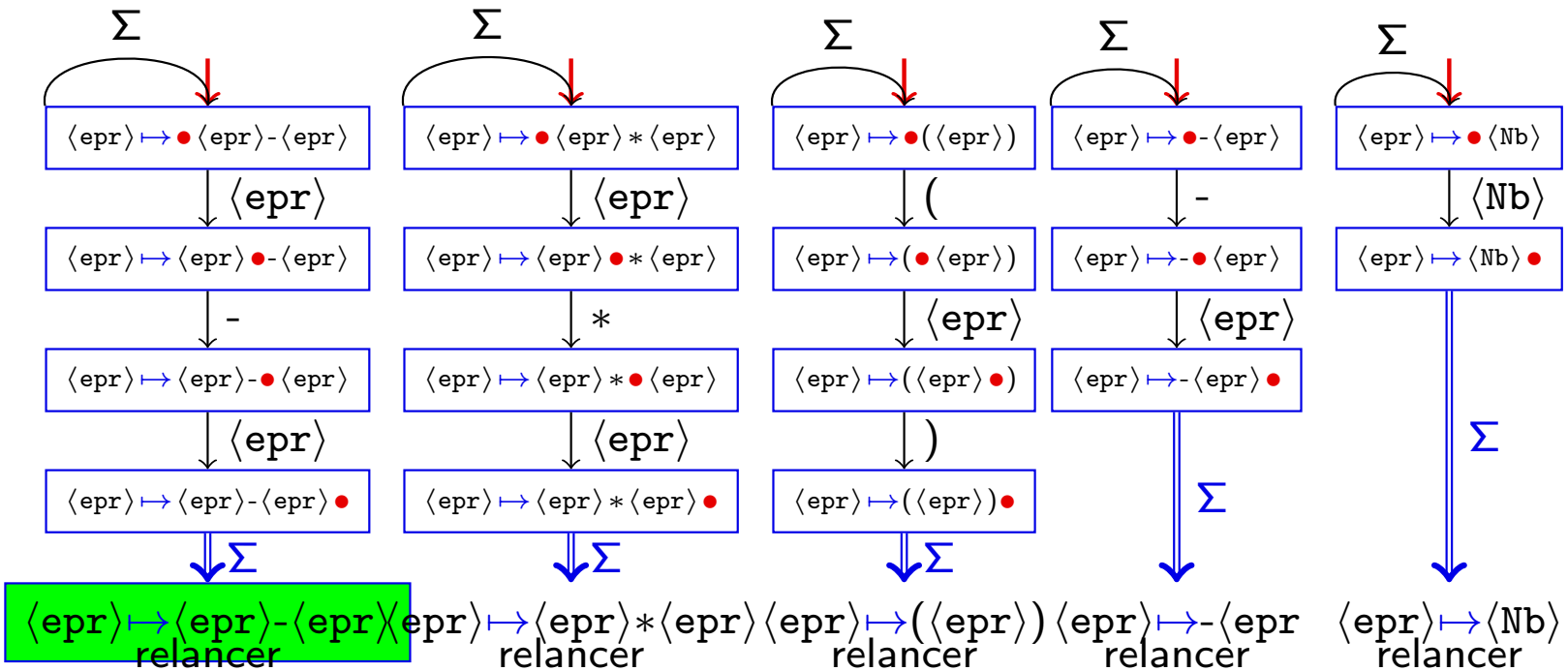


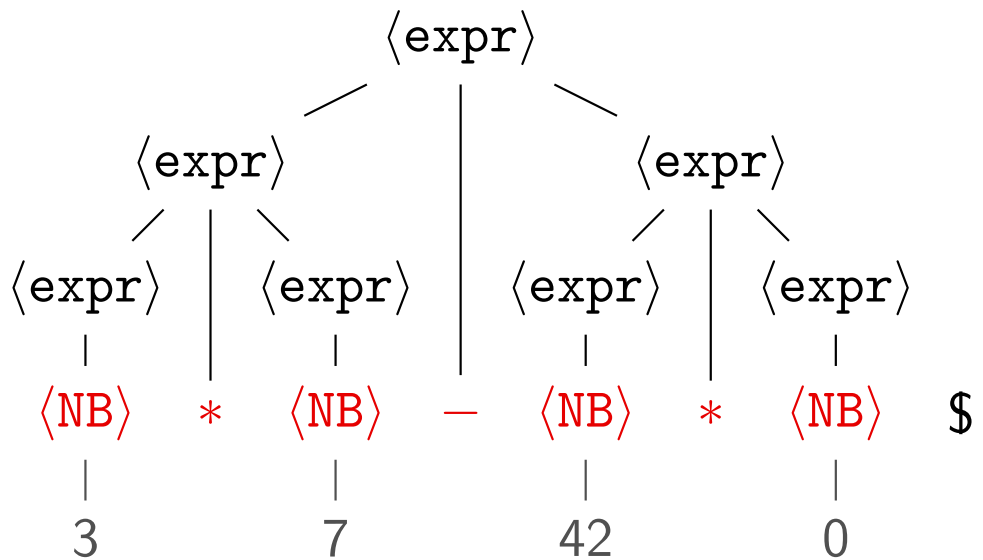
Automate du parseur LR en action



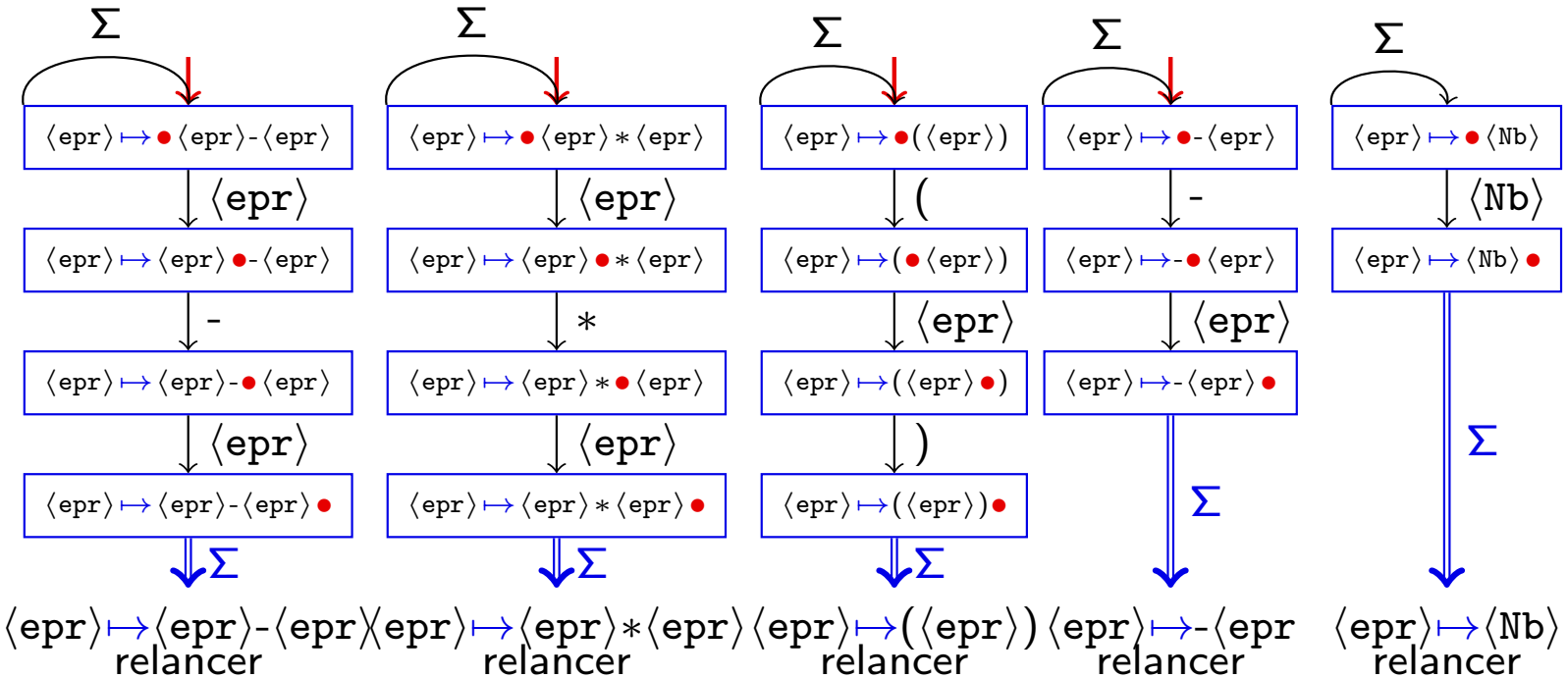


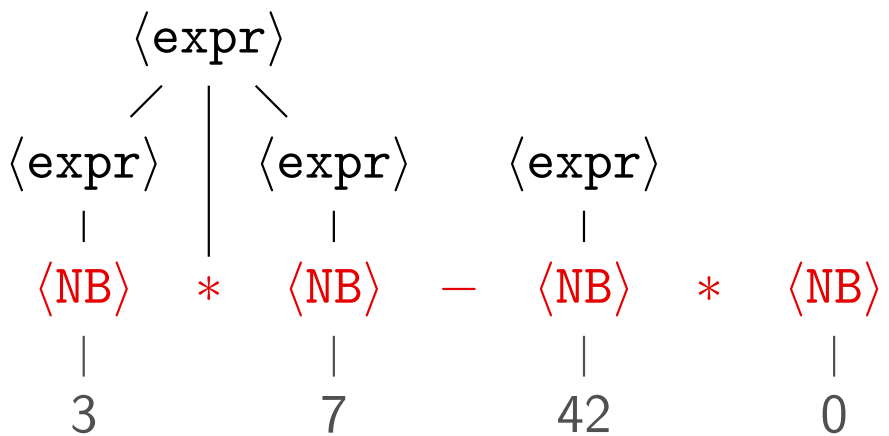
Automate du parseur LR en action



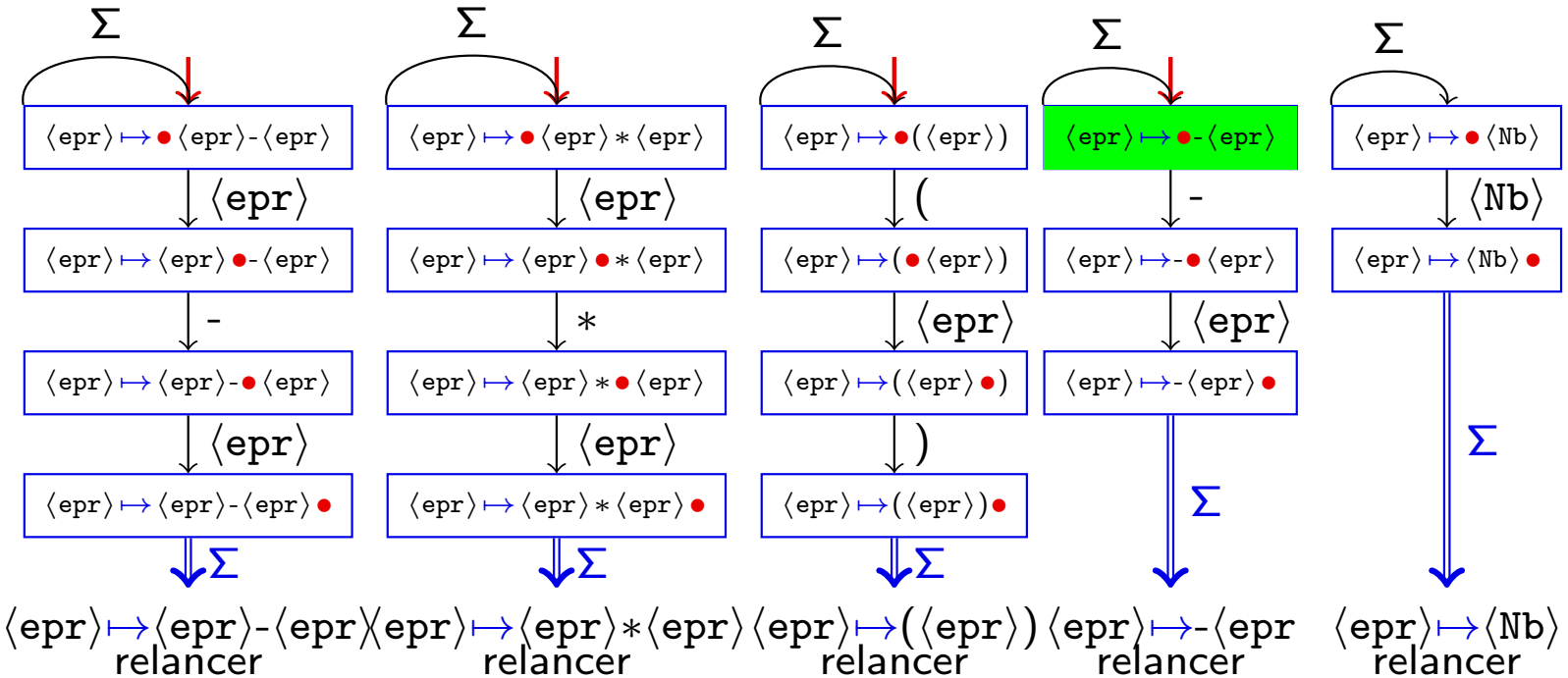


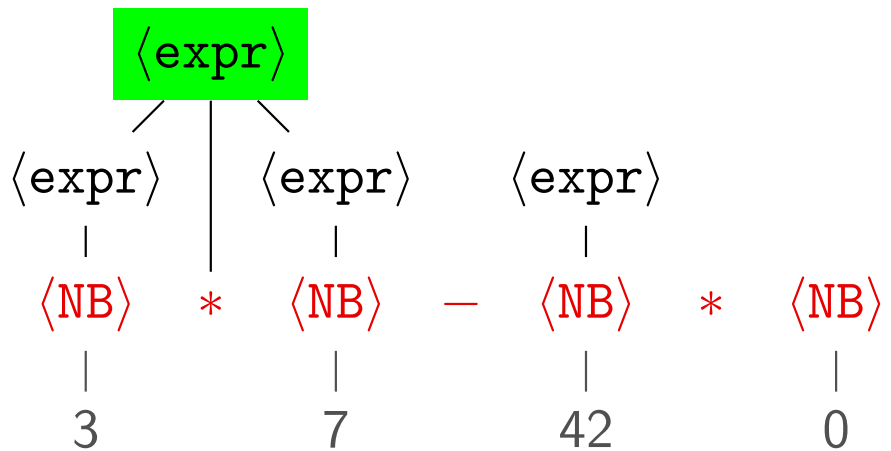
Échec de la tentative



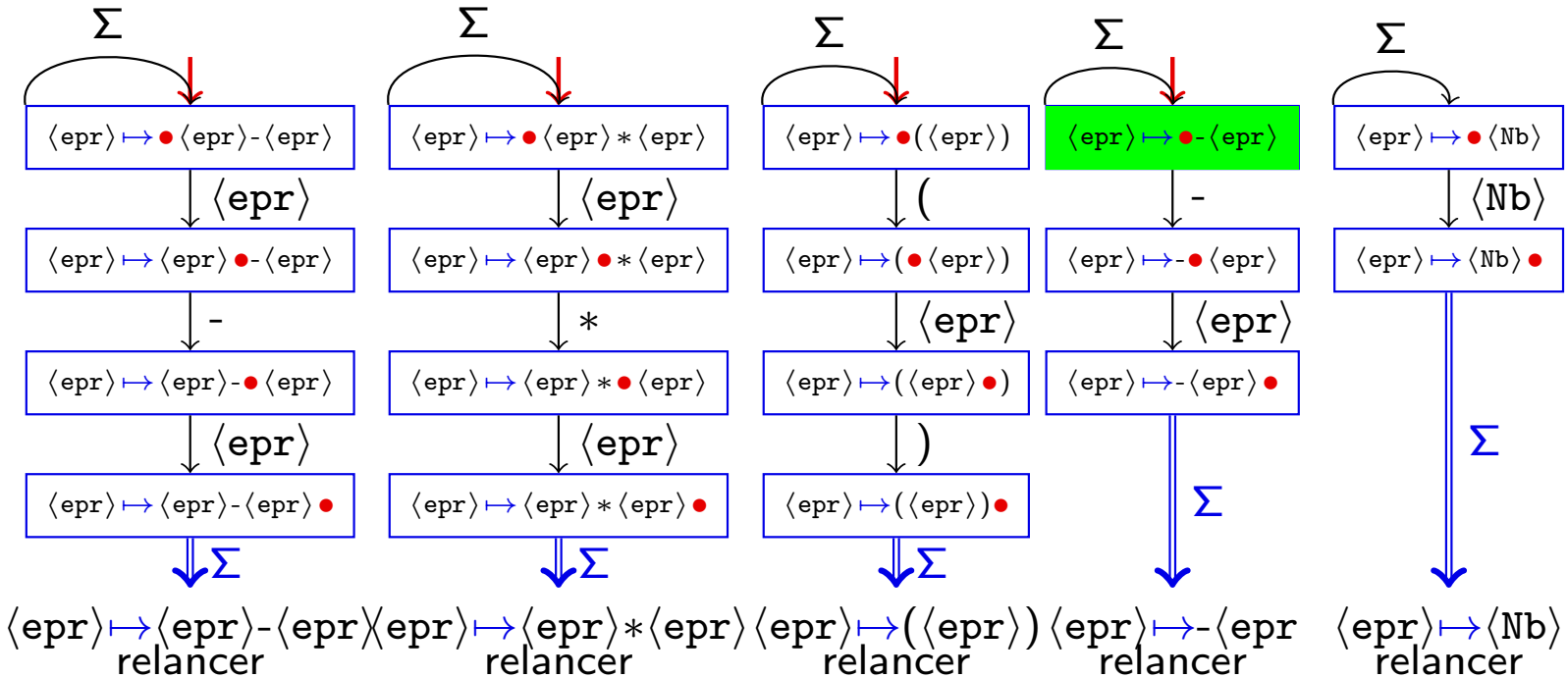


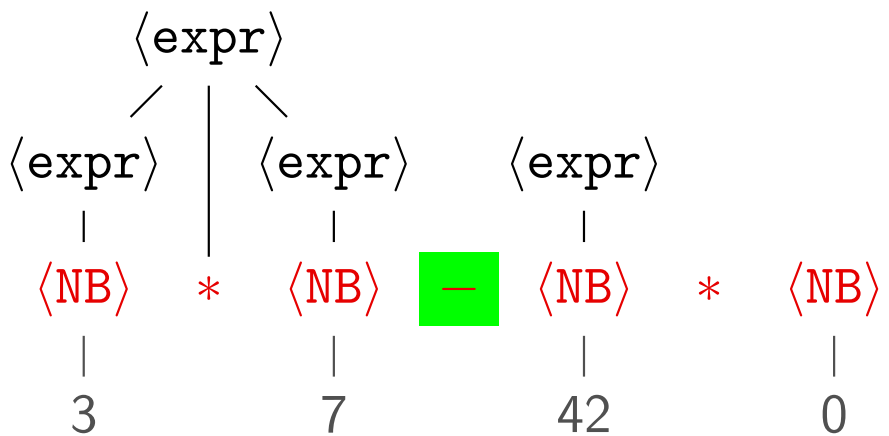
Échec de la tentative



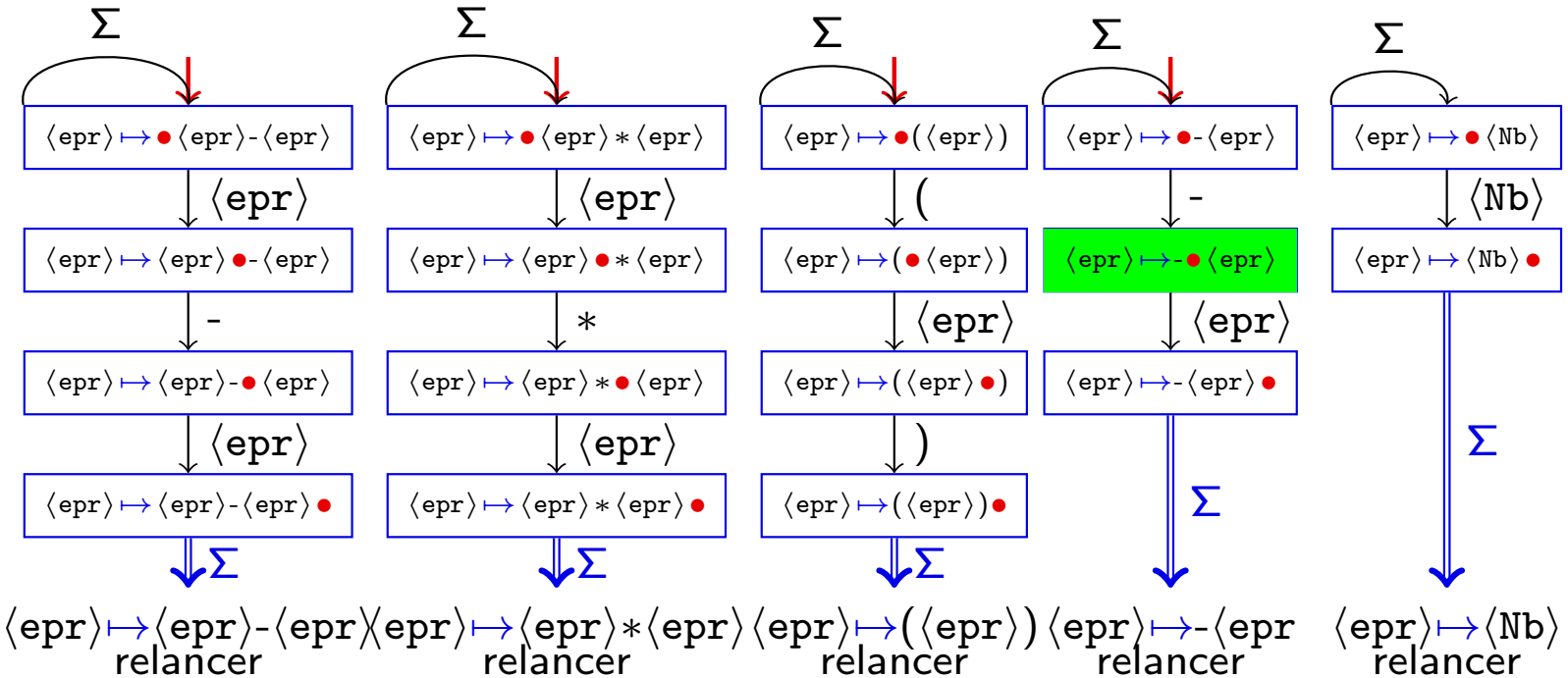


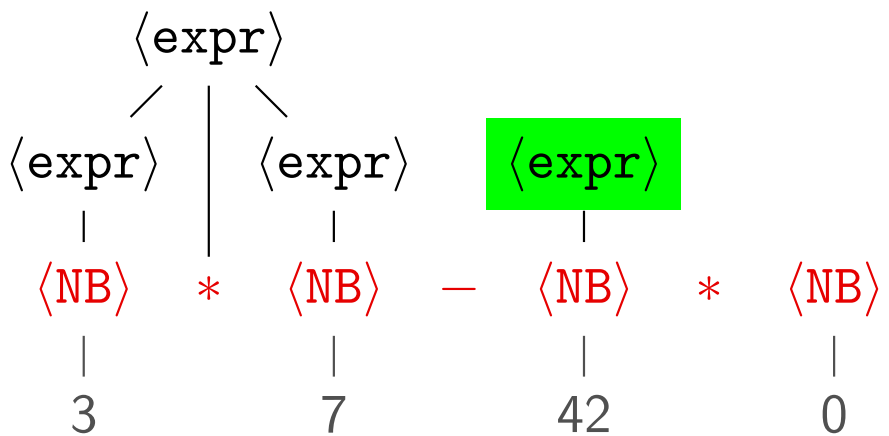
Échec de la tentative



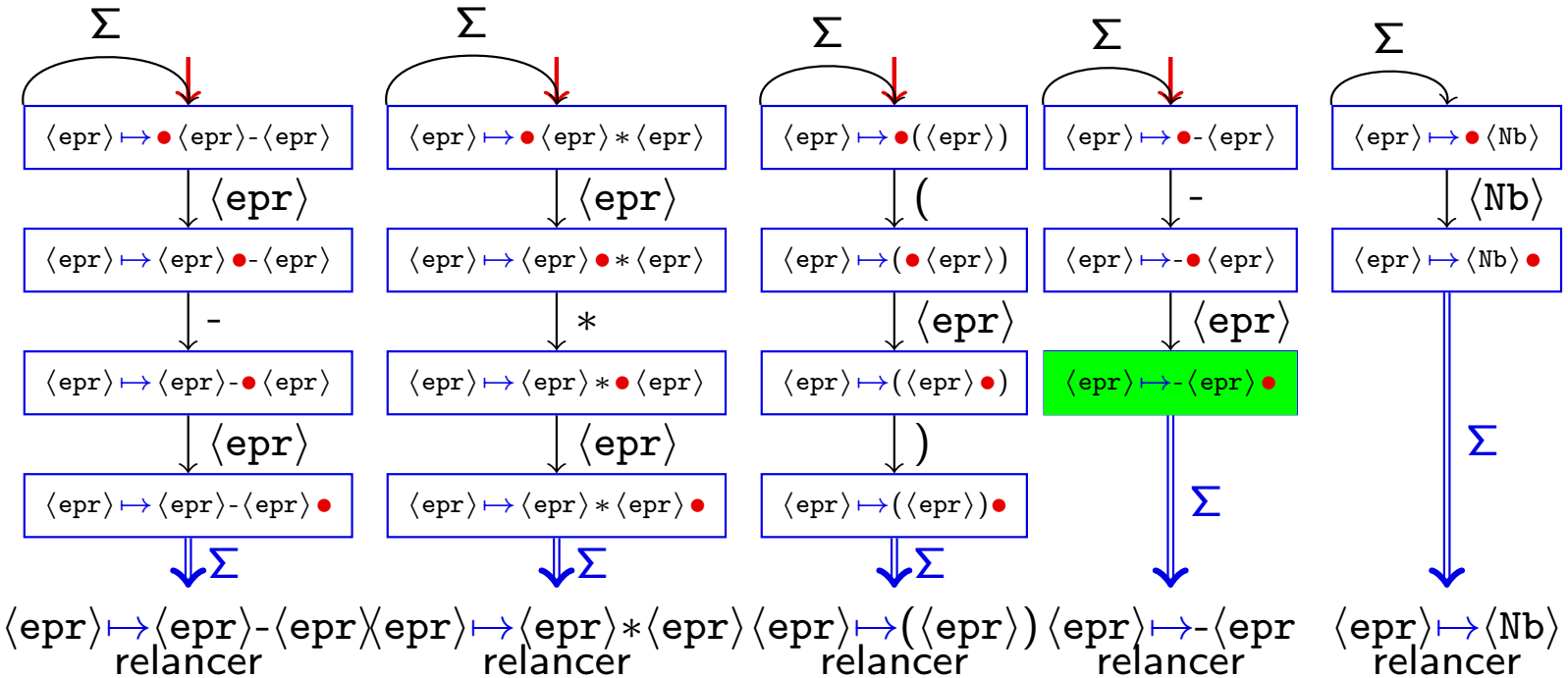


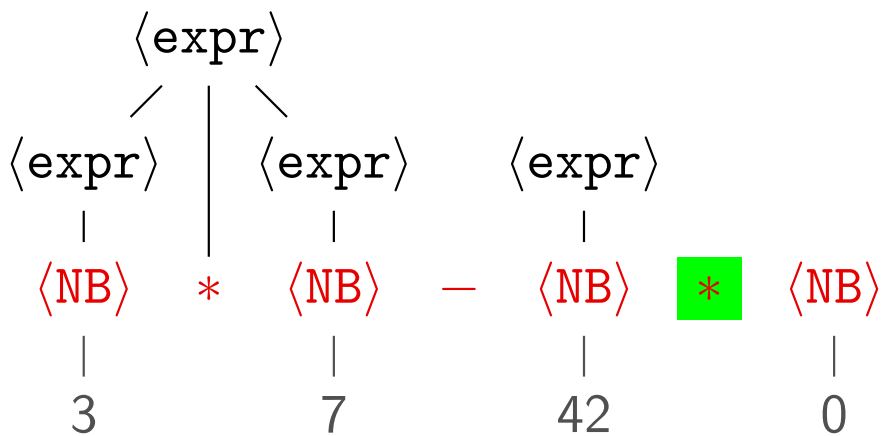
Échec de la tentative



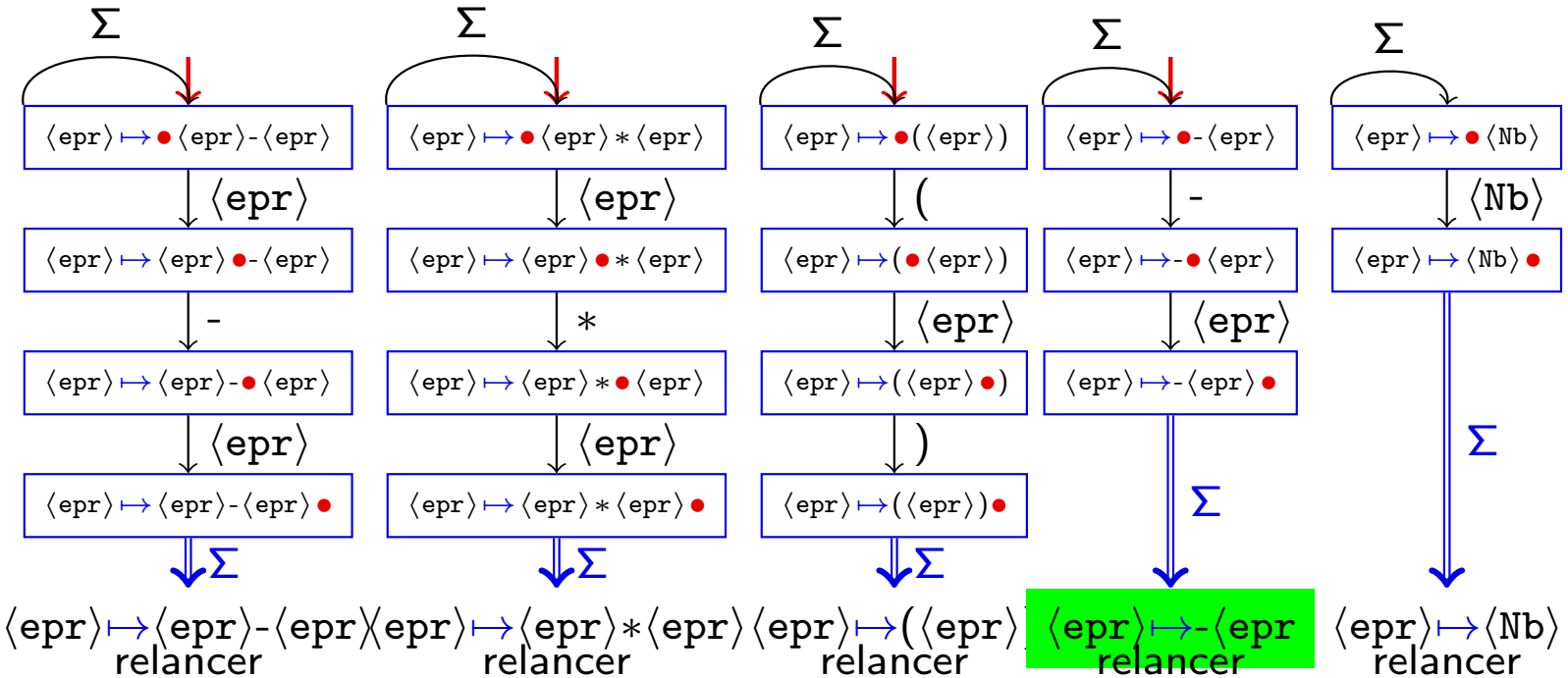


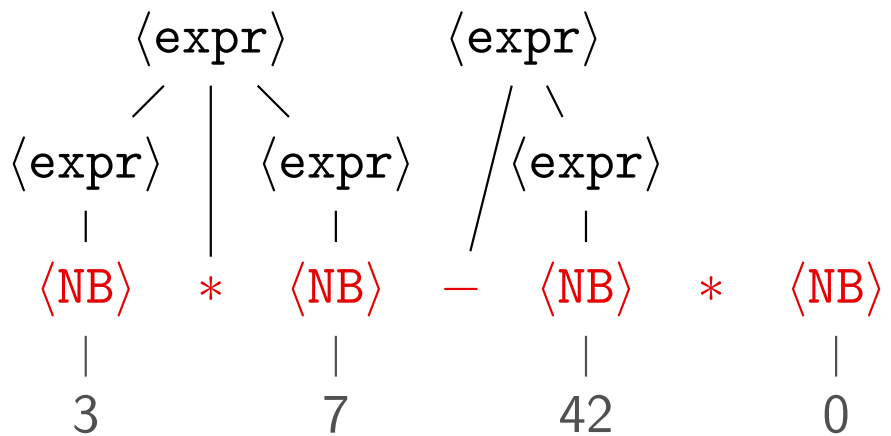
Échec de la tentative





Échec de la tentative

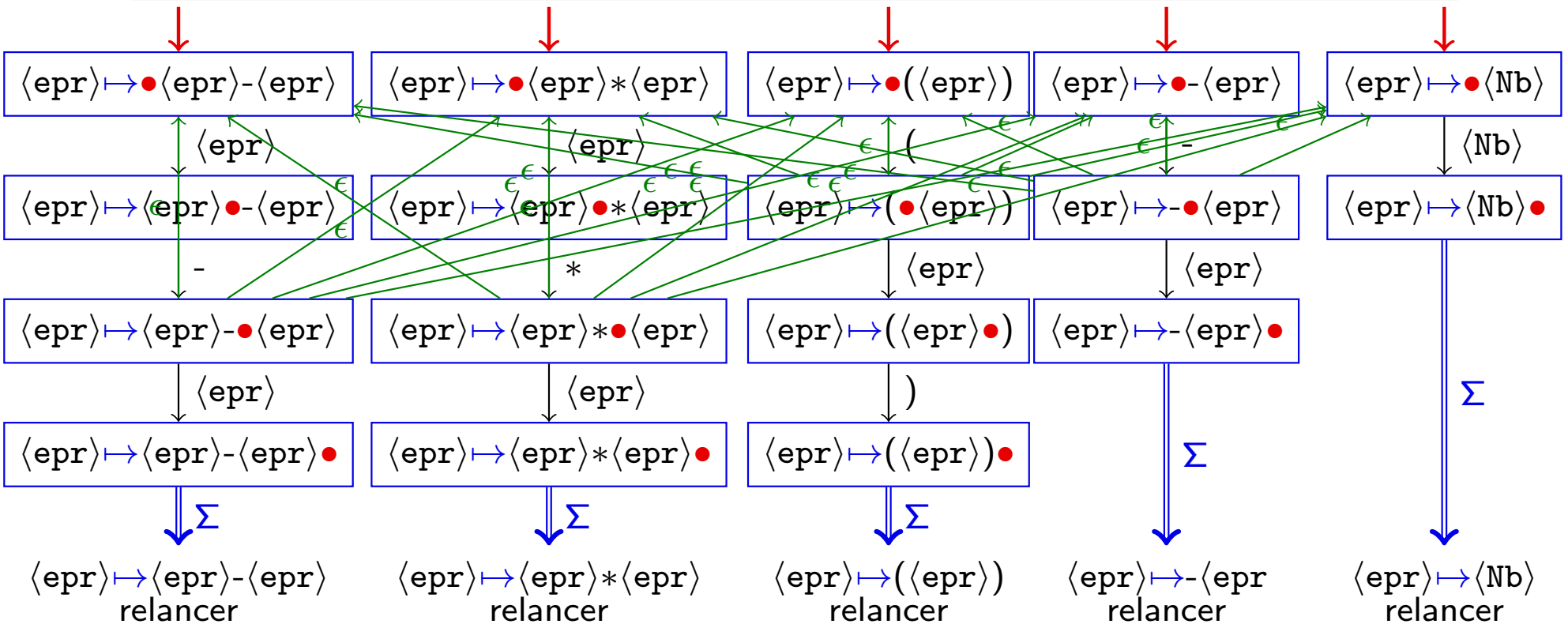




Seconde tentative

Idée : Introduire de la contextualité

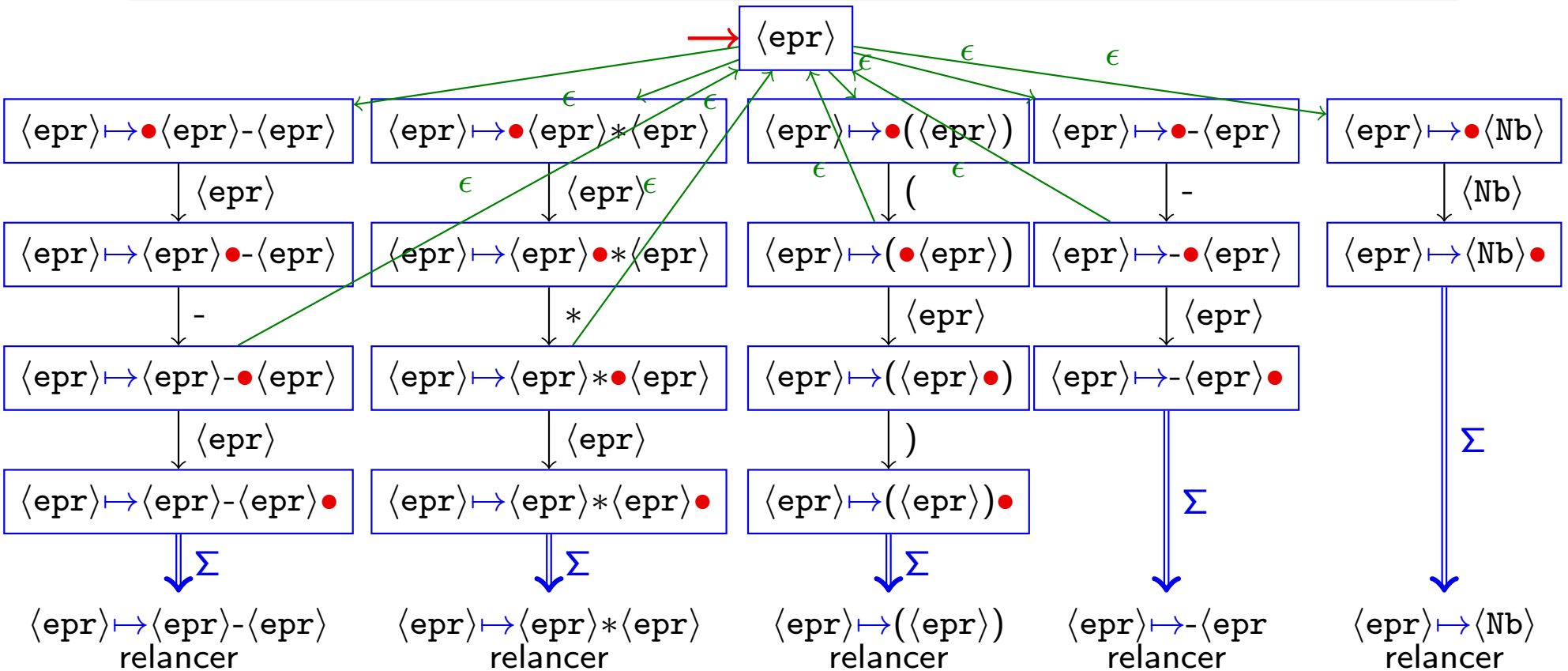
- On ne "teste" pas toutes les règles à tout moment,
- uniquement quand une autre règle peut avoir besoin du NT,
- en utilisant des ϵ-transitions



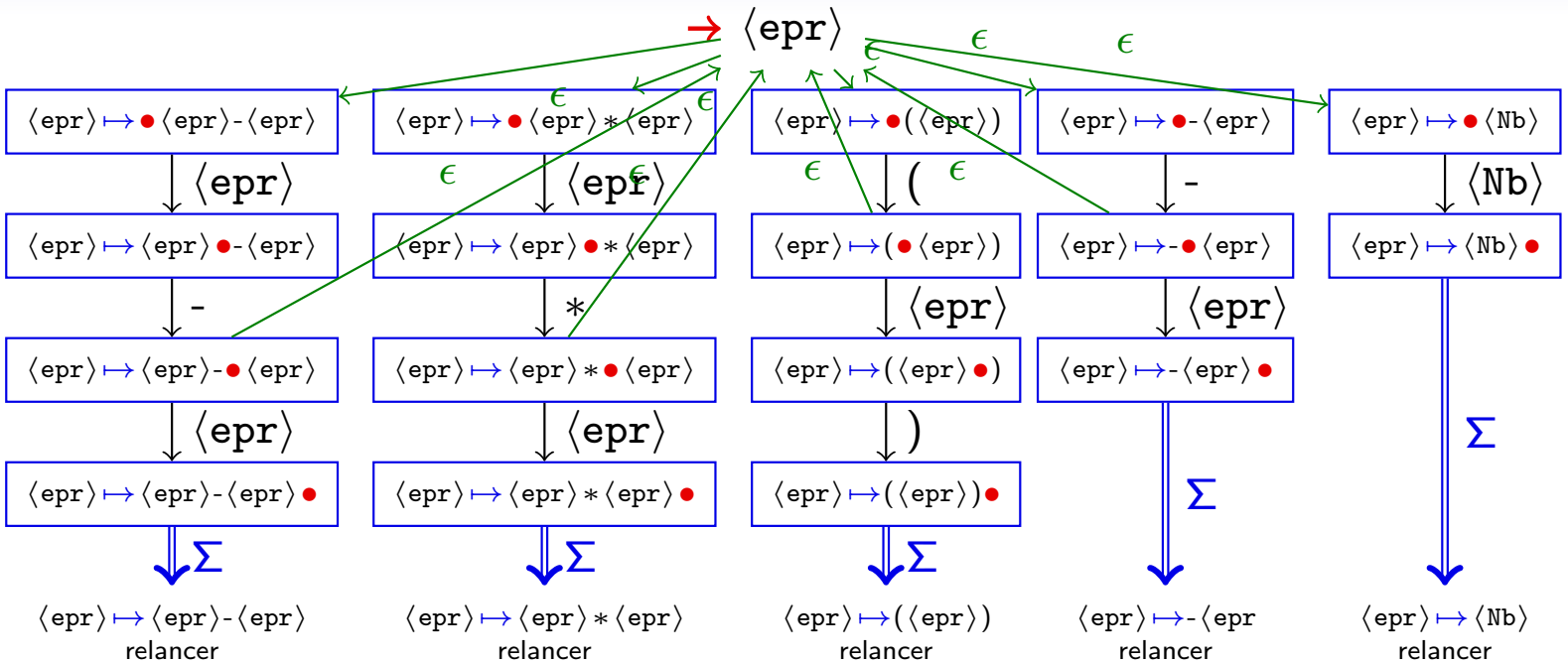
Seconde tentative

Idée : Introduire de la contextualité

- On ne "teste" pas toutes les règles à tout moment,
- uniquement quand une autre règle peut avoir besoin du NT,
- en utilisant des ϵ-transitions



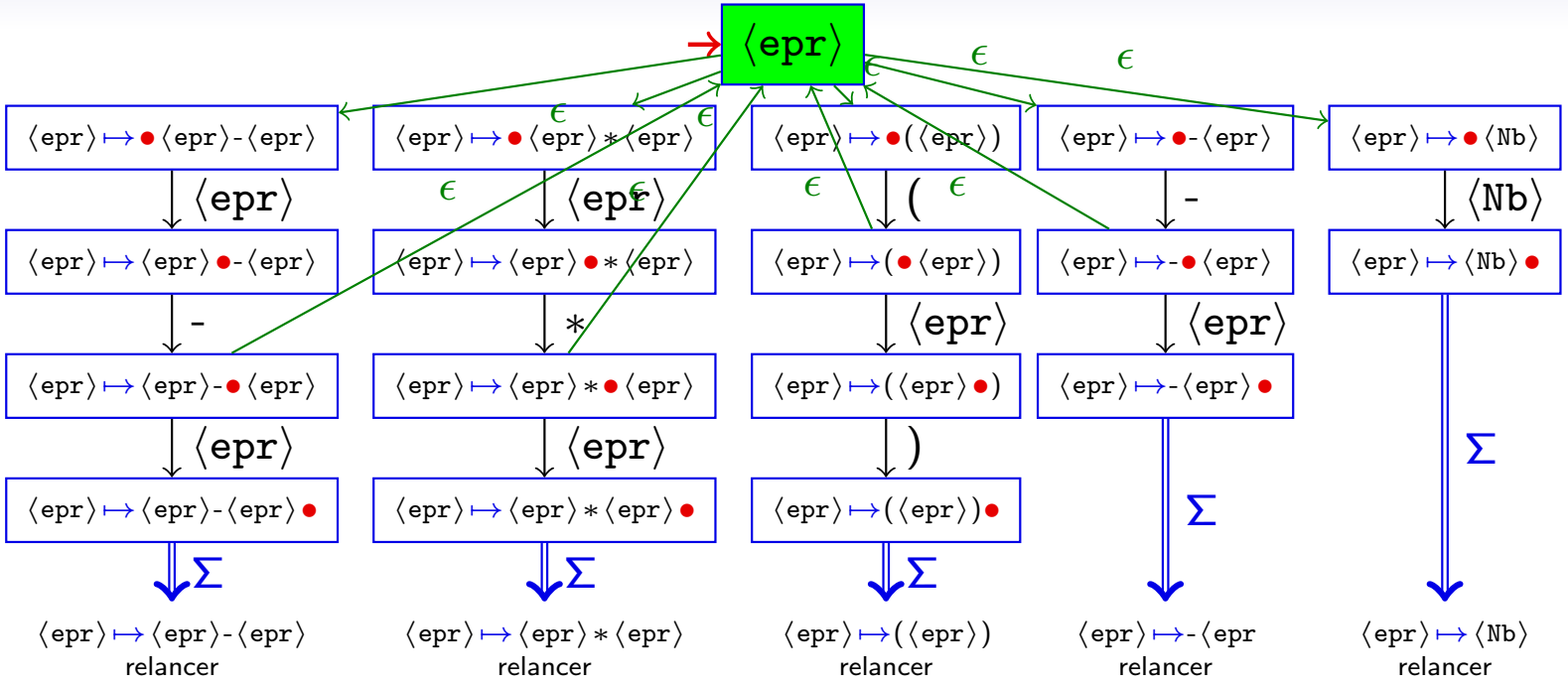
Seconde tentative en action



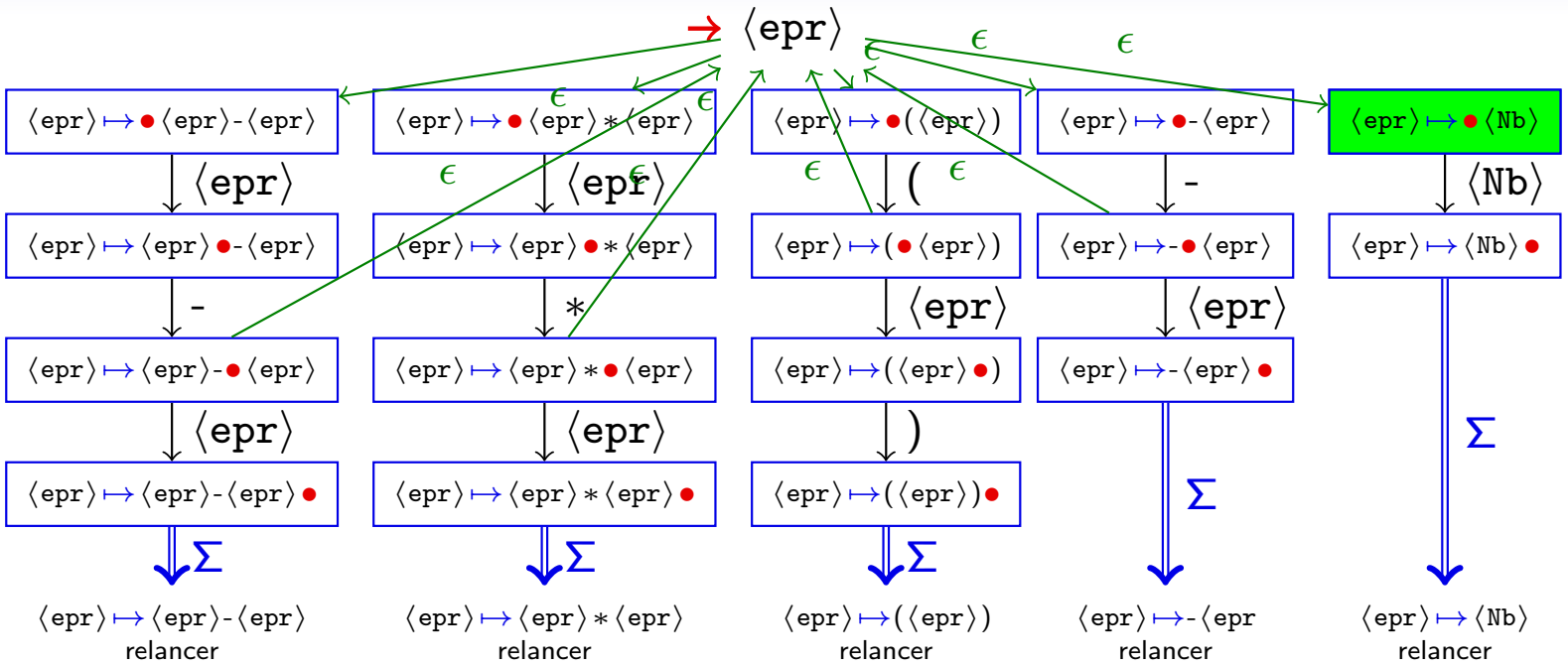
⟨NB⟩ ∗ ⟨NB⟩ − ⟨NB⟩ ∗ ⟨NB⟩ $

3 7 42 0

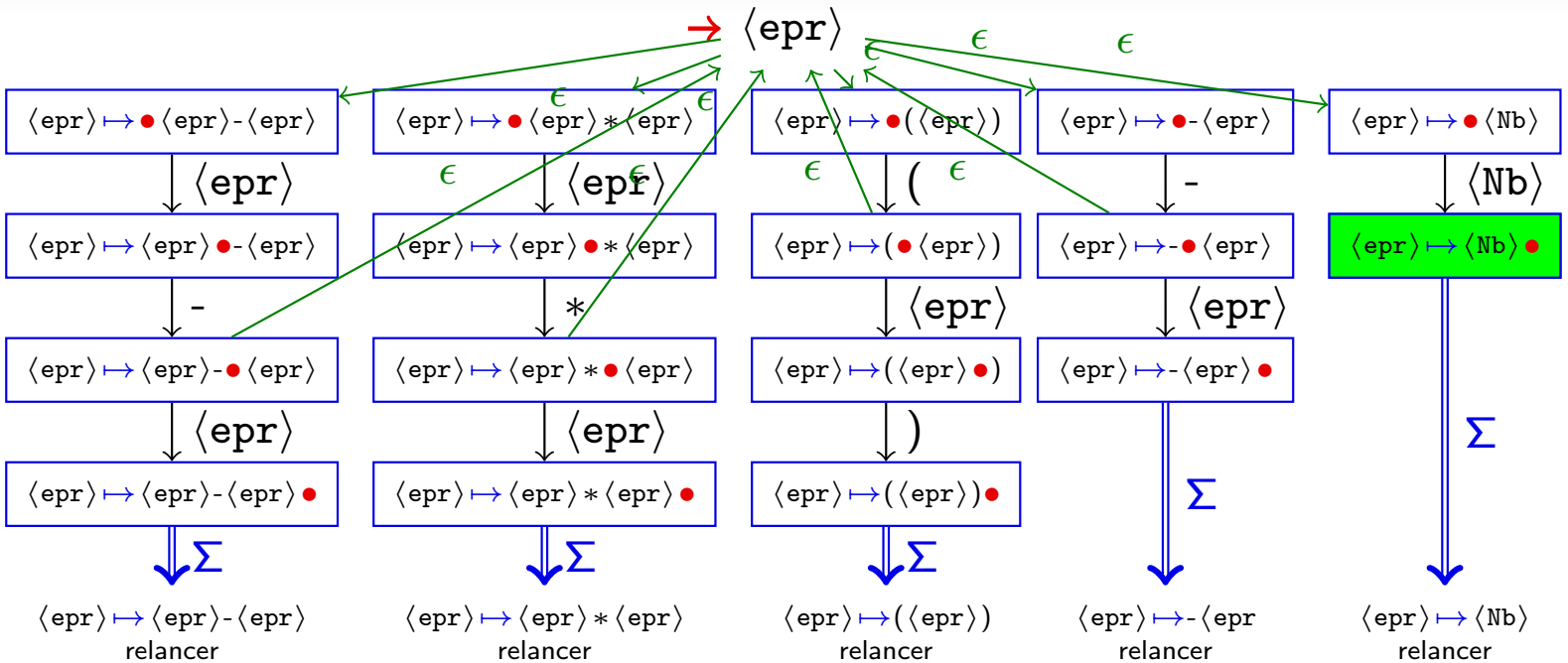
Seconde tentative en action

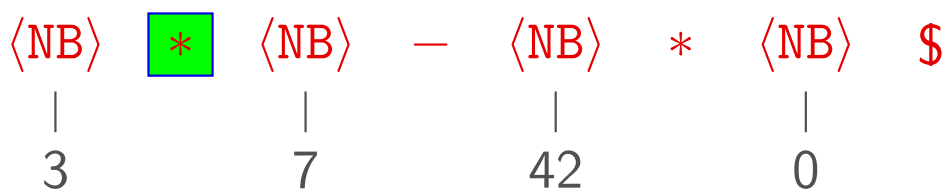


Seconde tentative en action

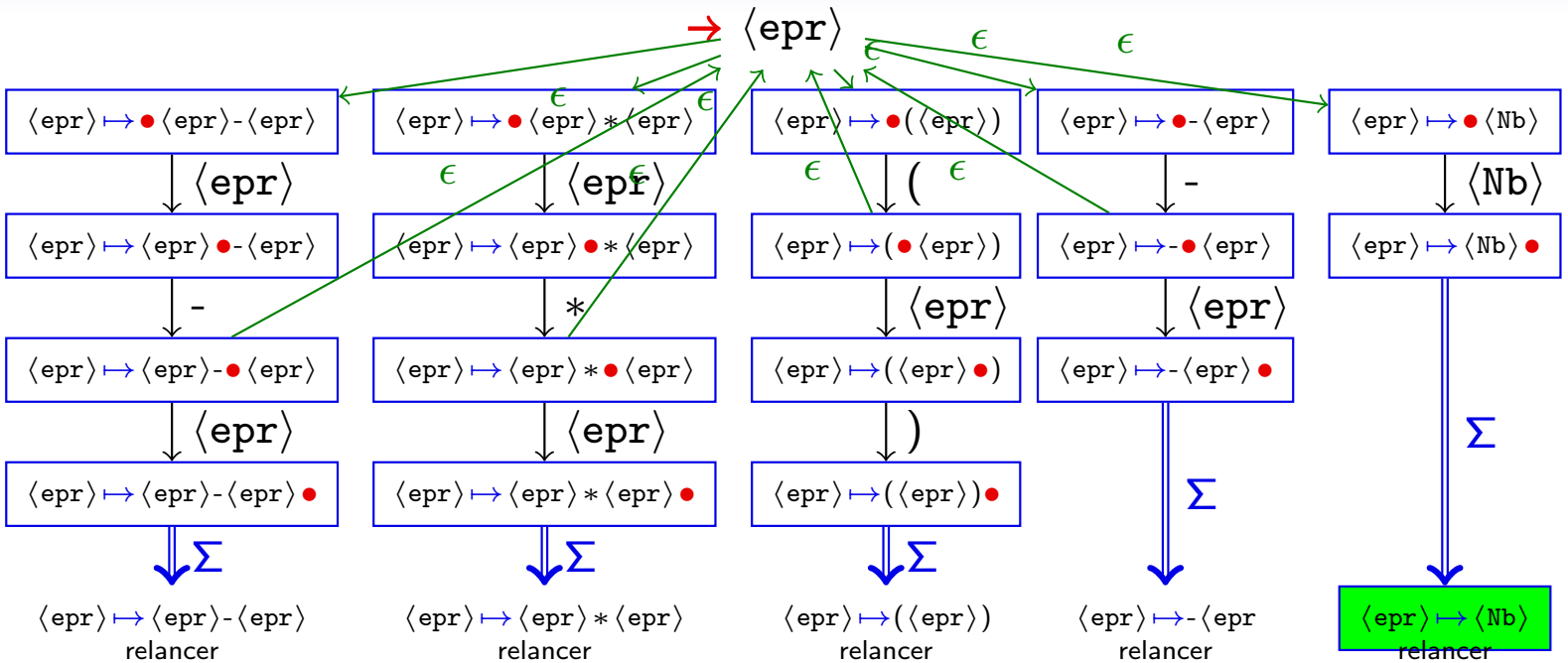


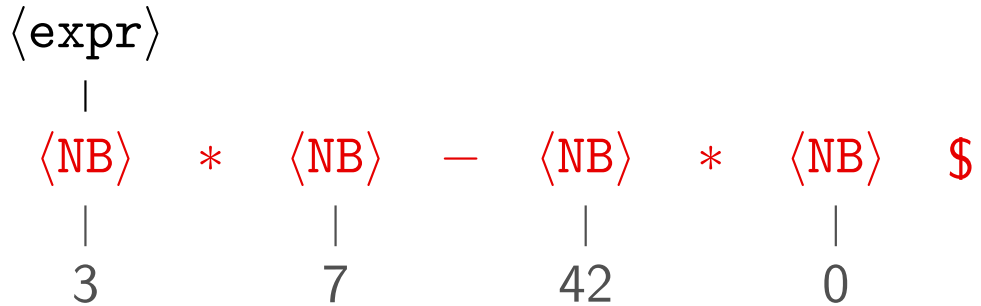
Seconde tentative en action



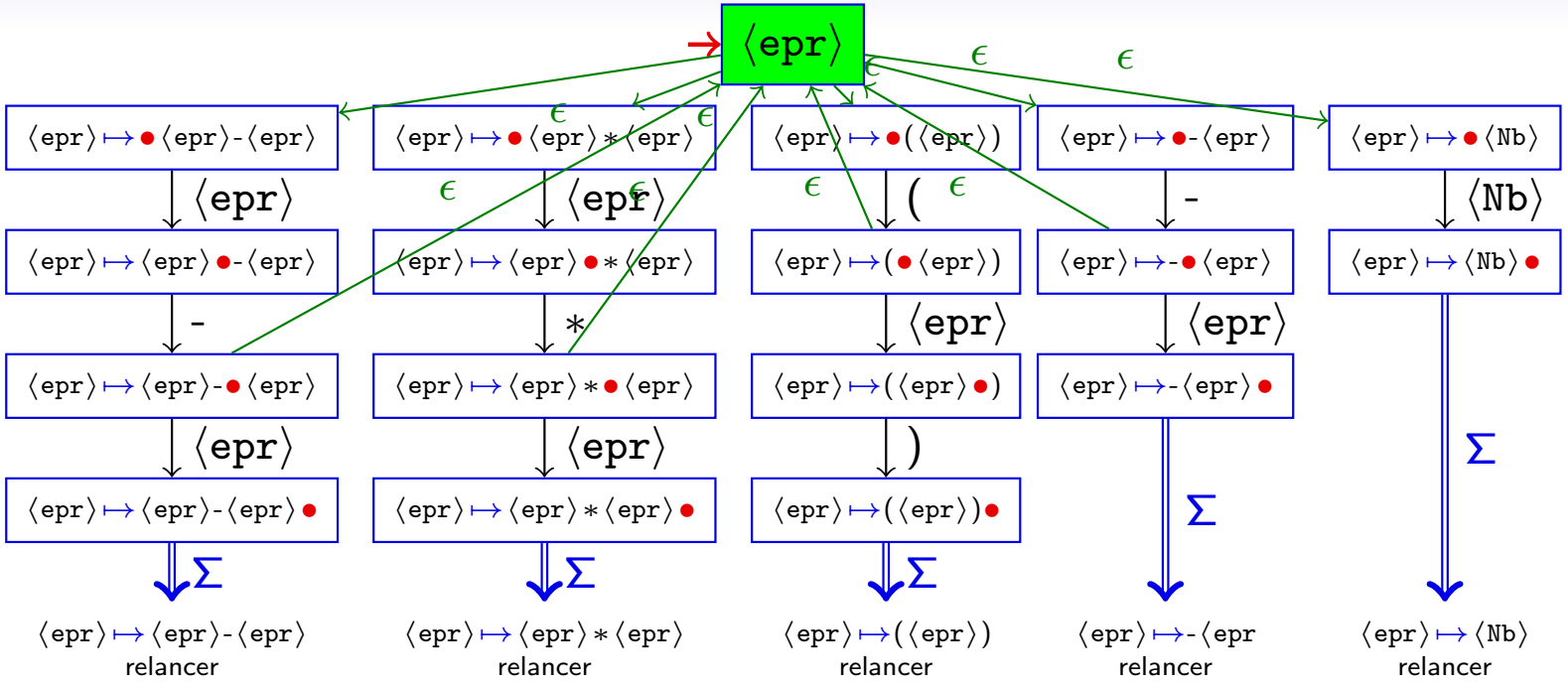


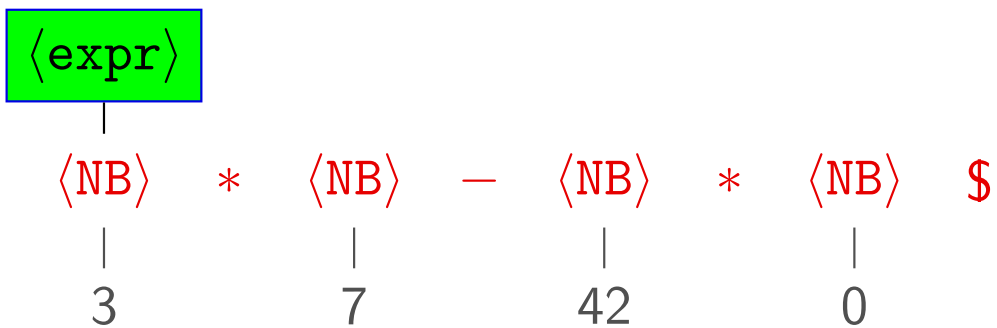
Seconde tentative en action



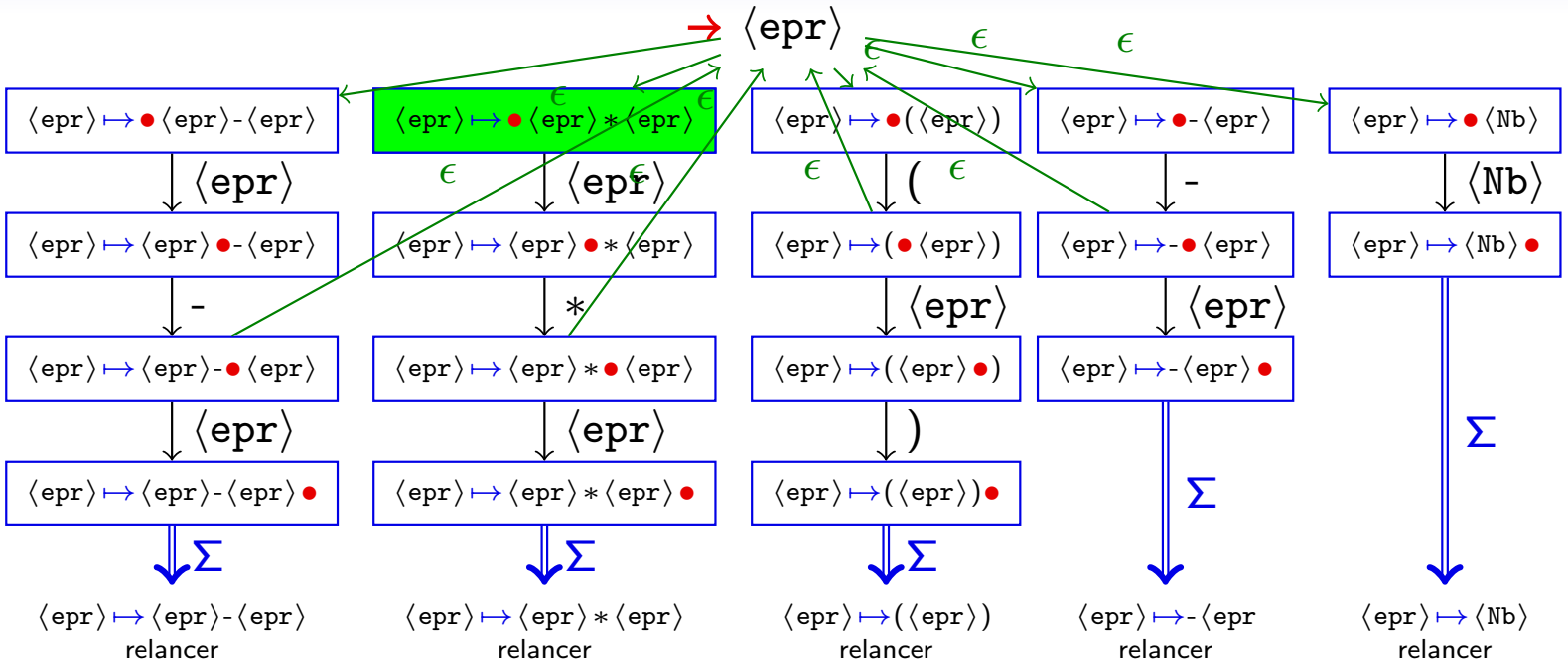


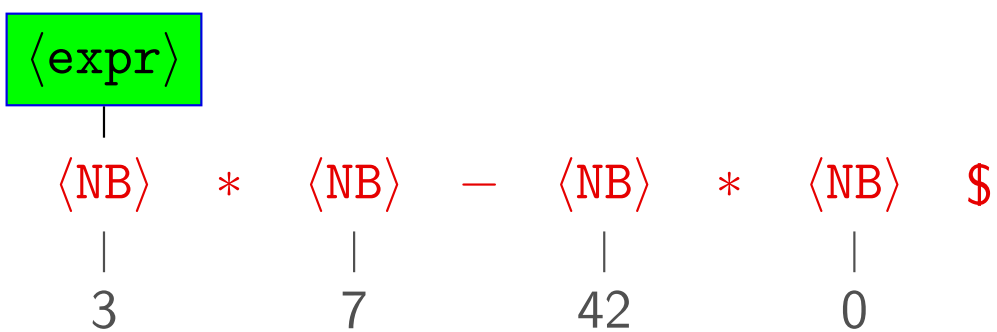
Seconde tentative en action



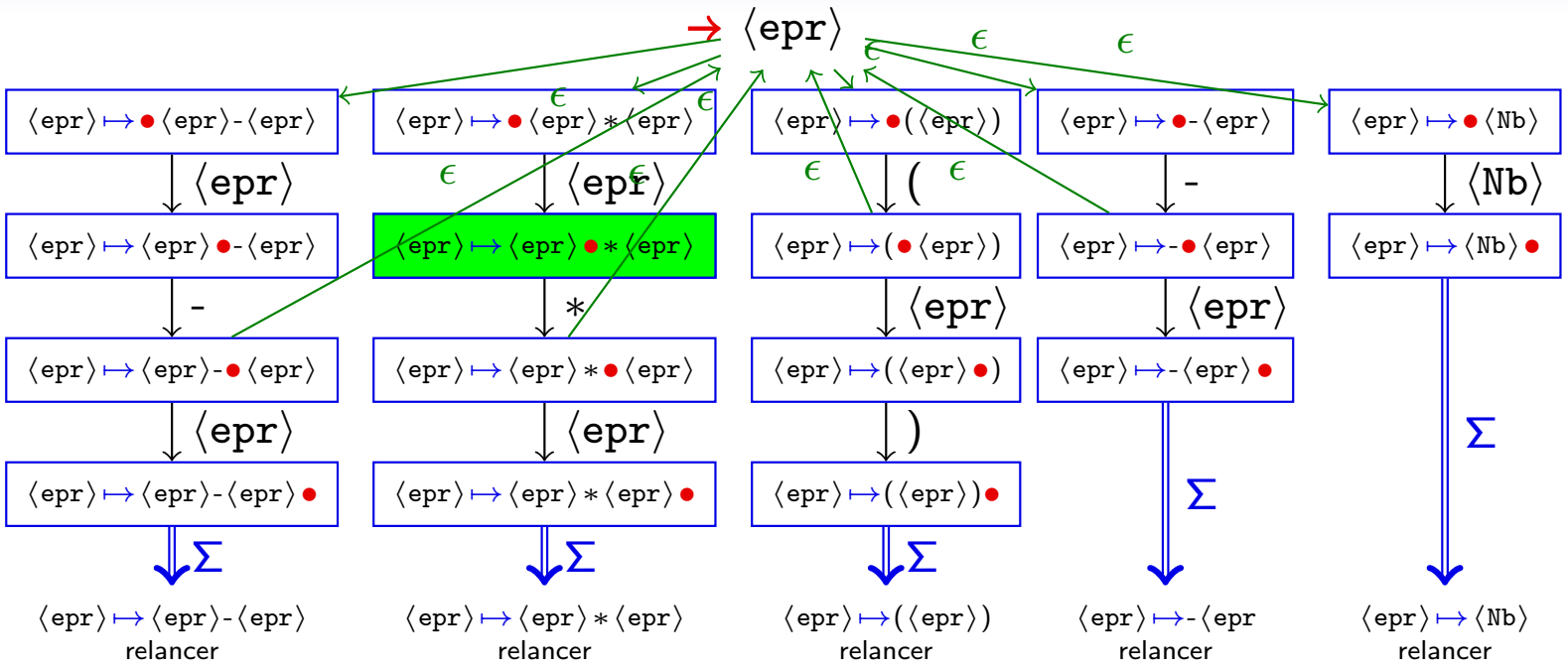


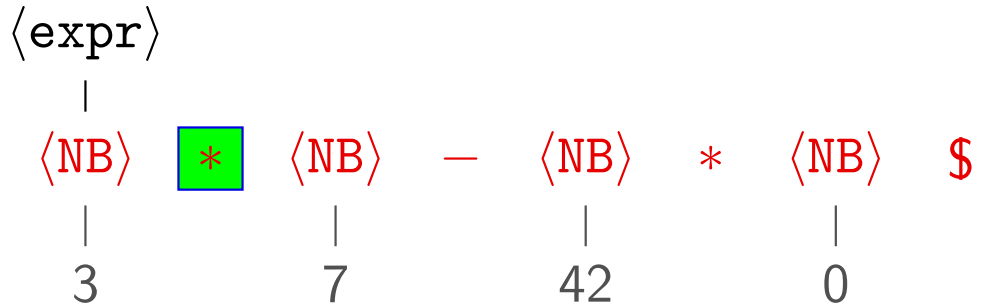
Seconde tentative en action



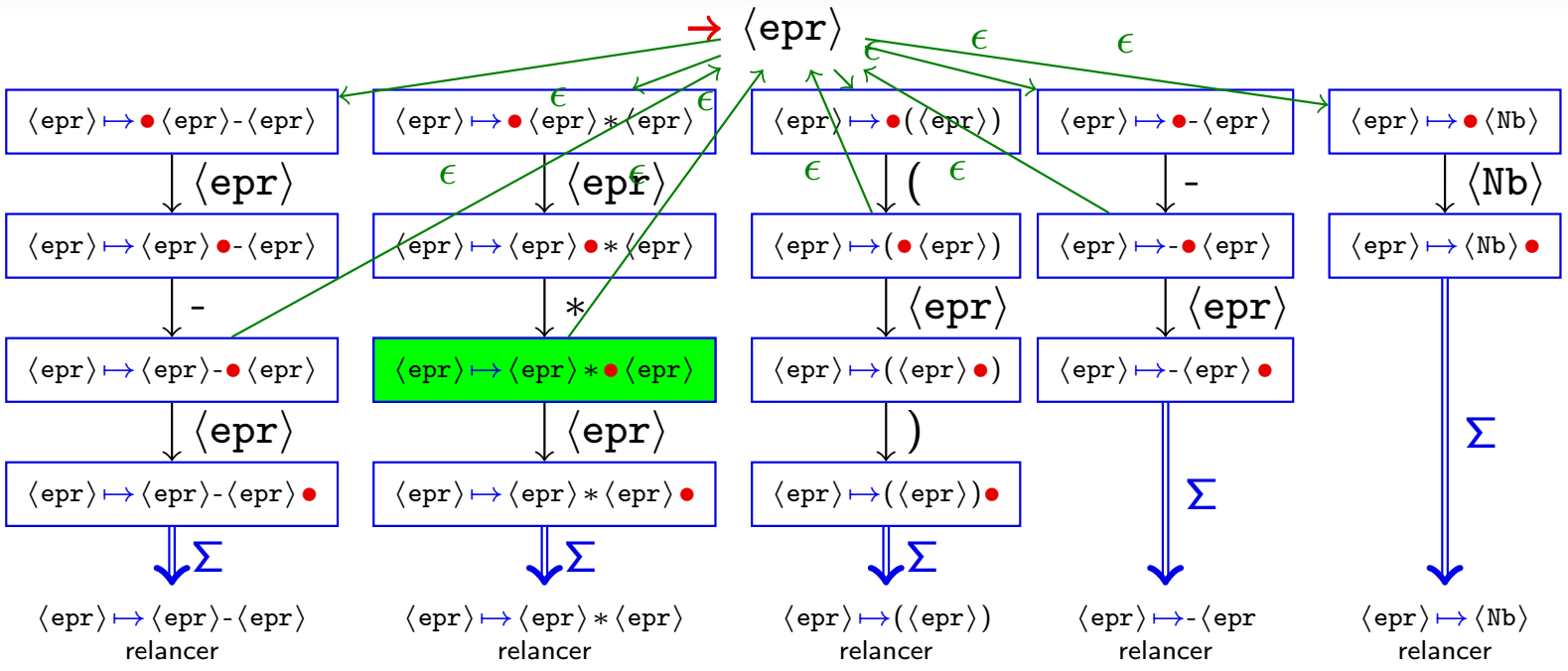


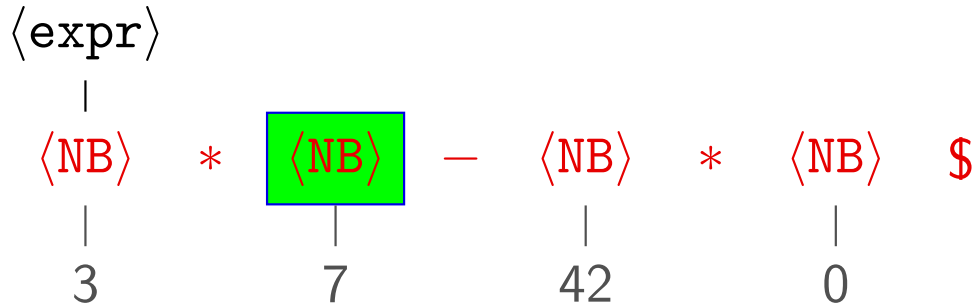
Seconde tentative en action



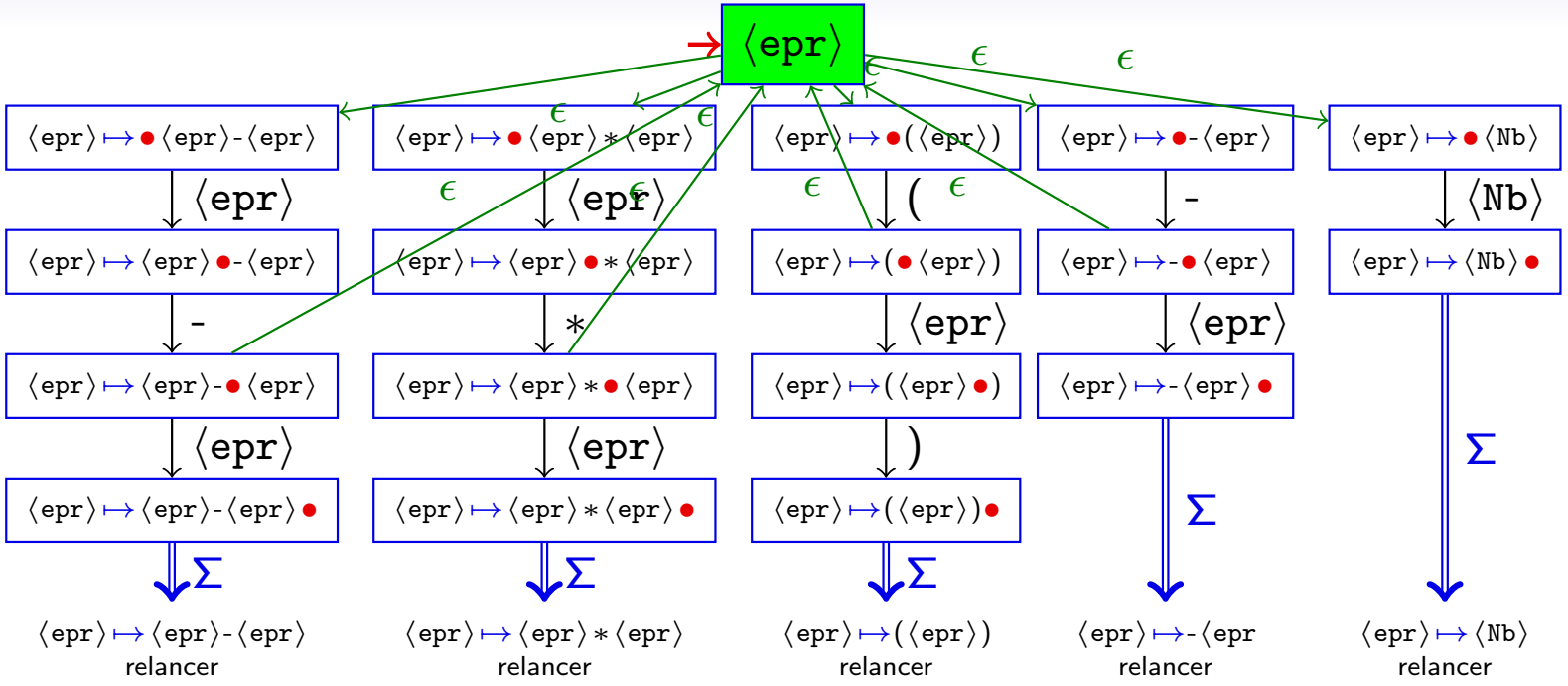


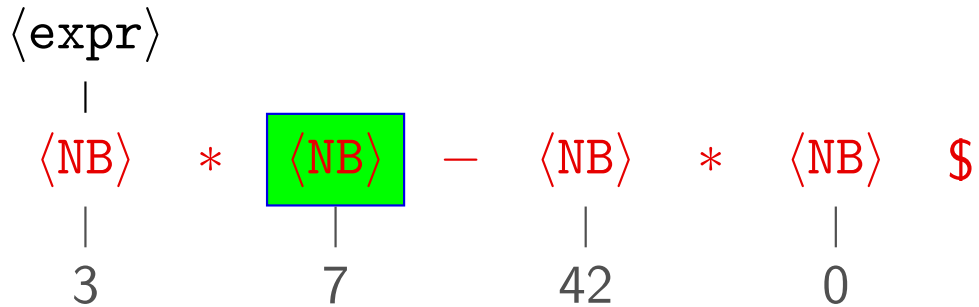
Seconde tentative en action



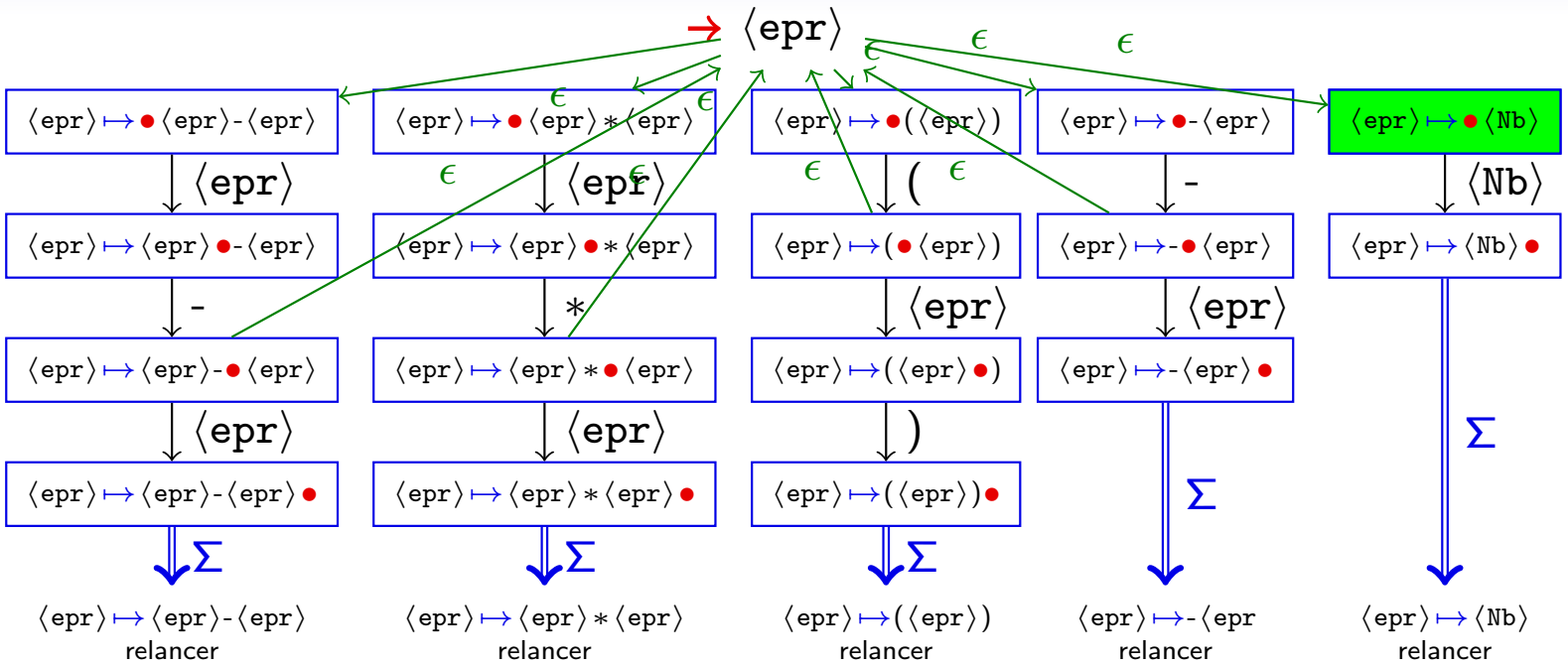


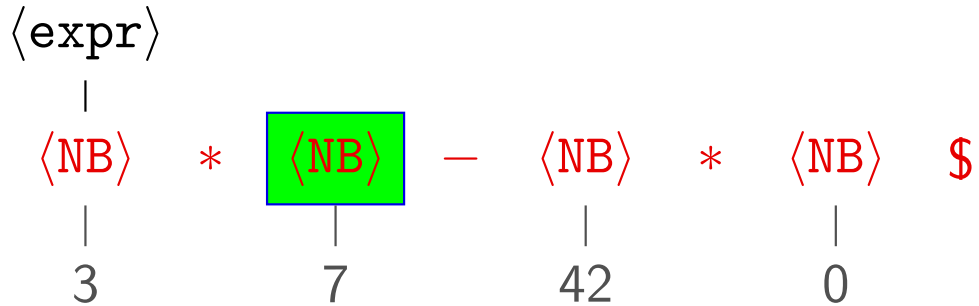
Seconde tentative en action



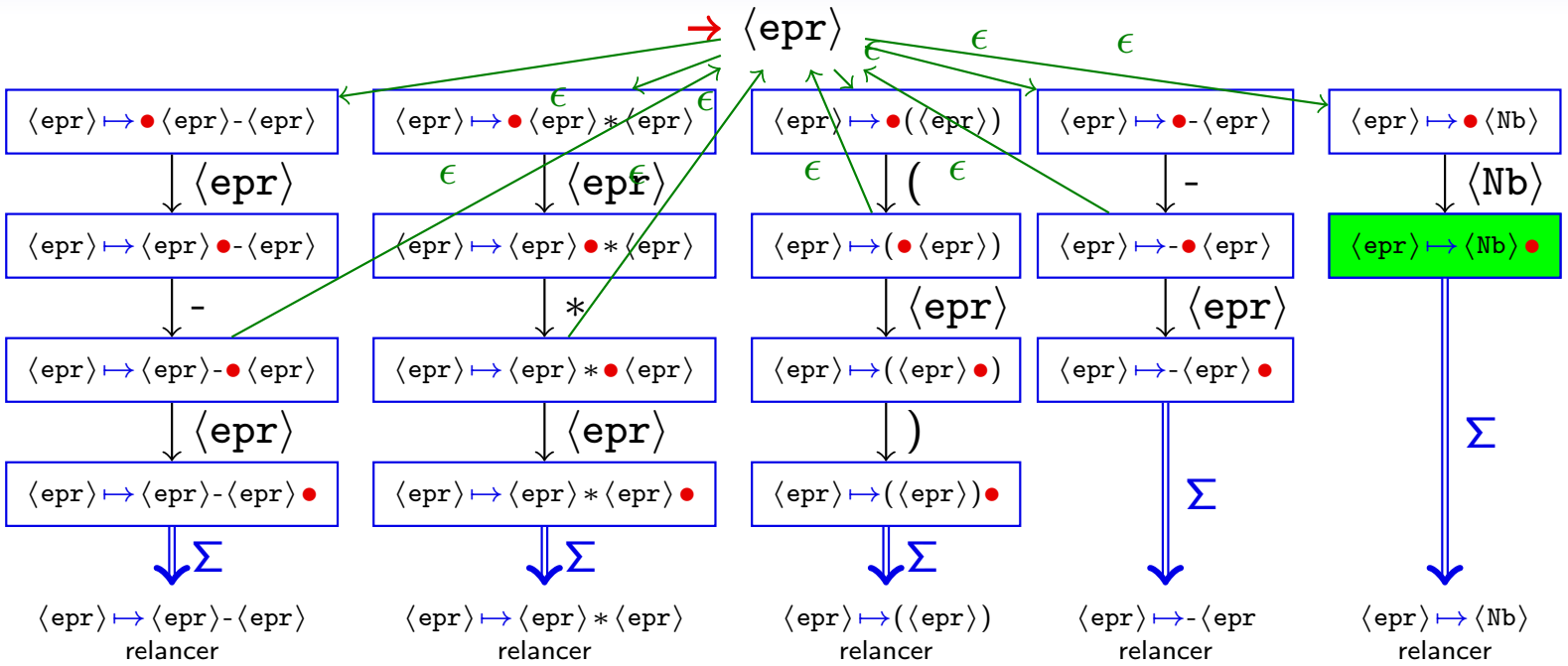


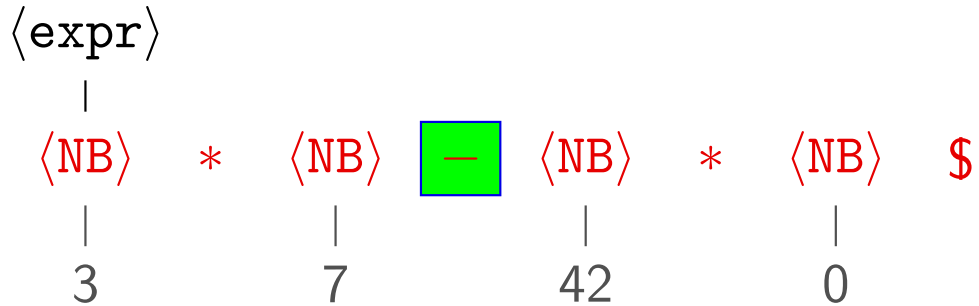
Seconde tentative en action



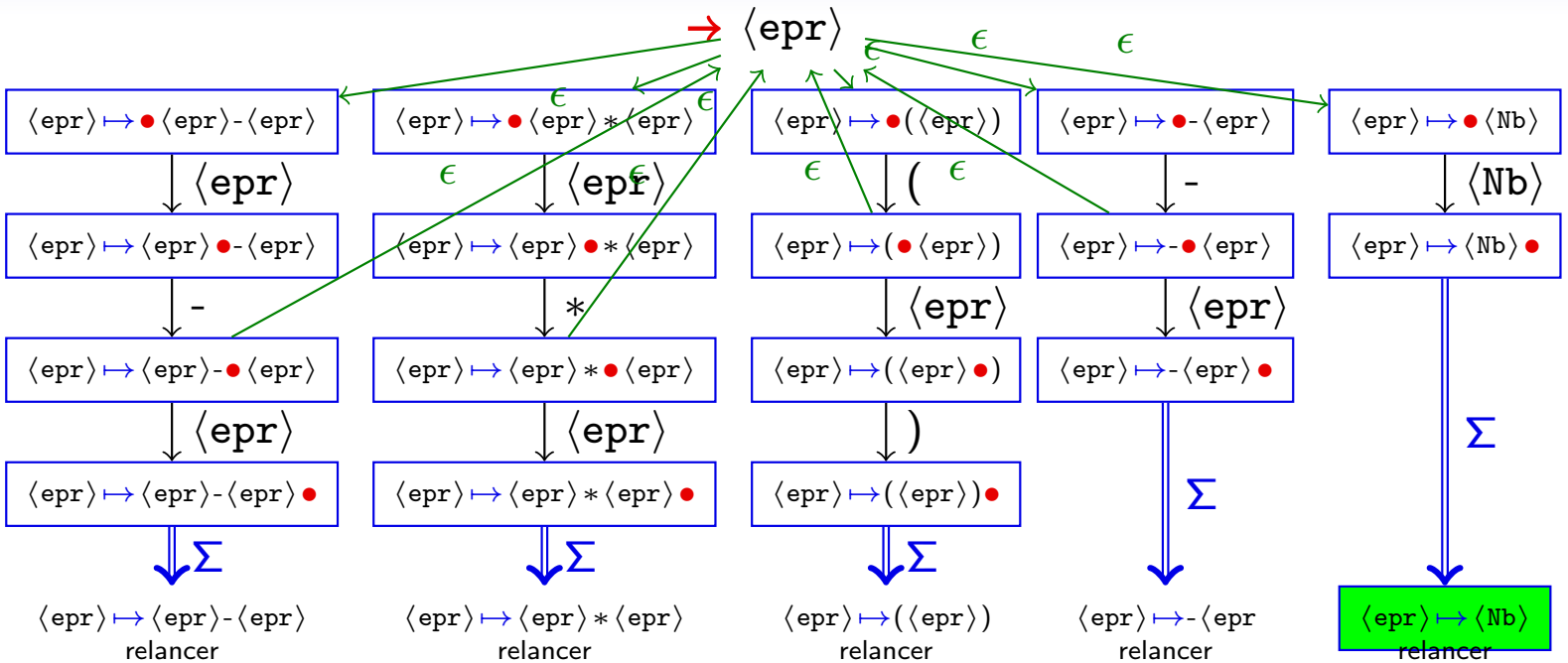


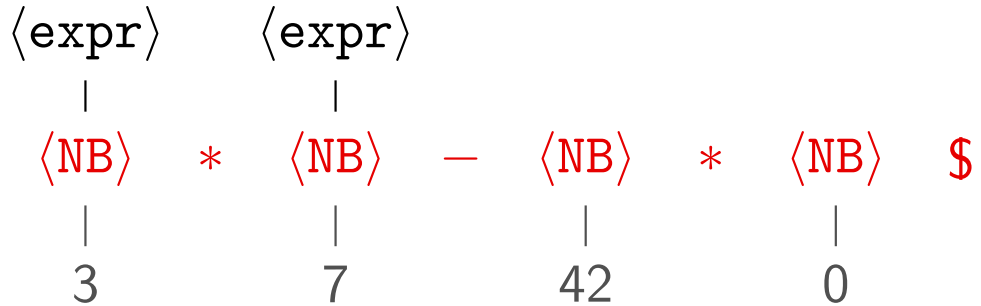
Seconde tentative en action



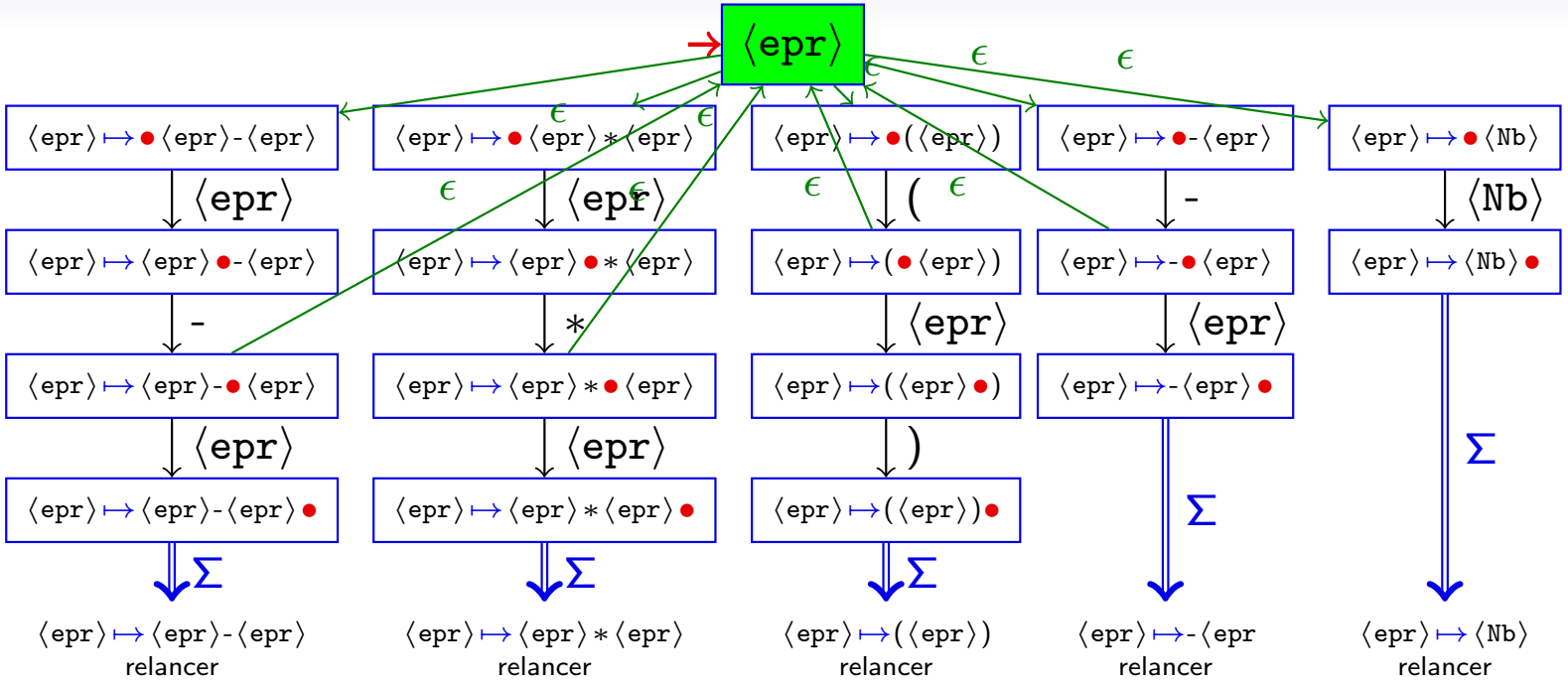


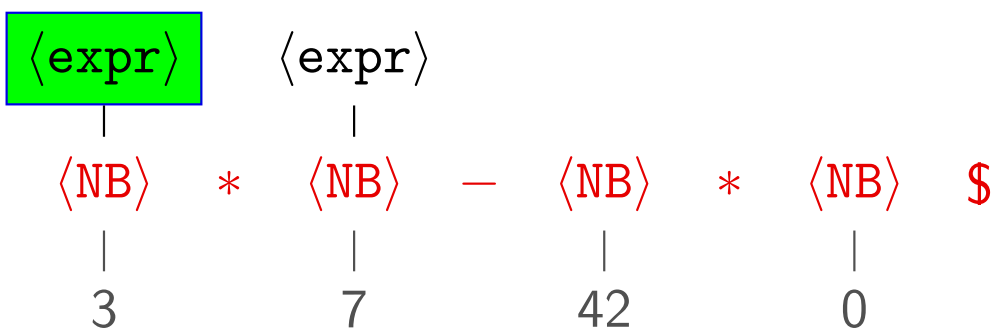
Seconde tentative en action



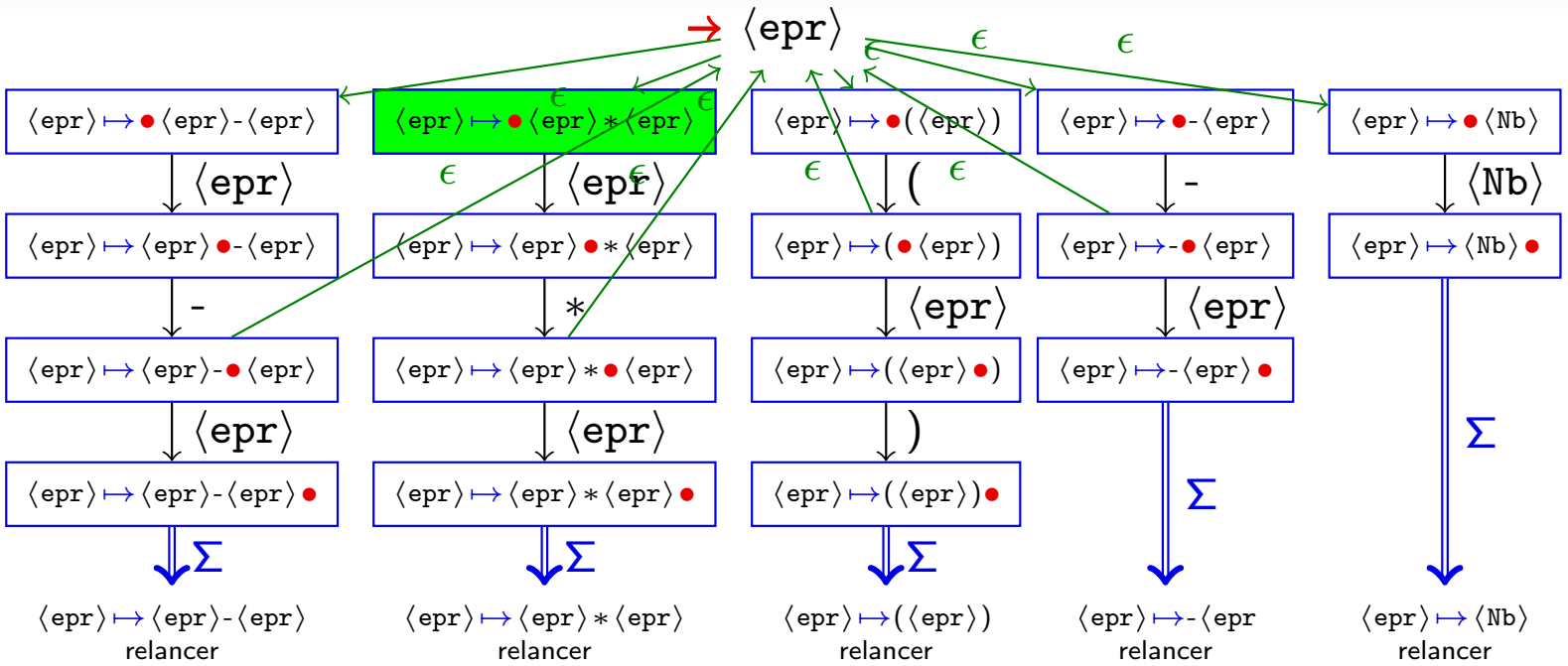


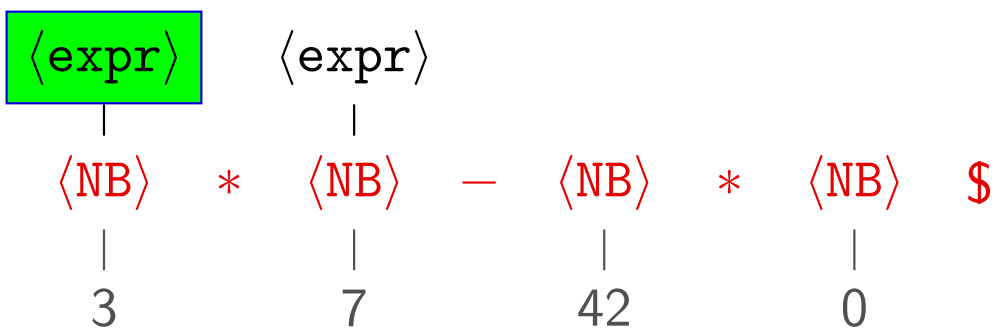
Seconde tentative en action



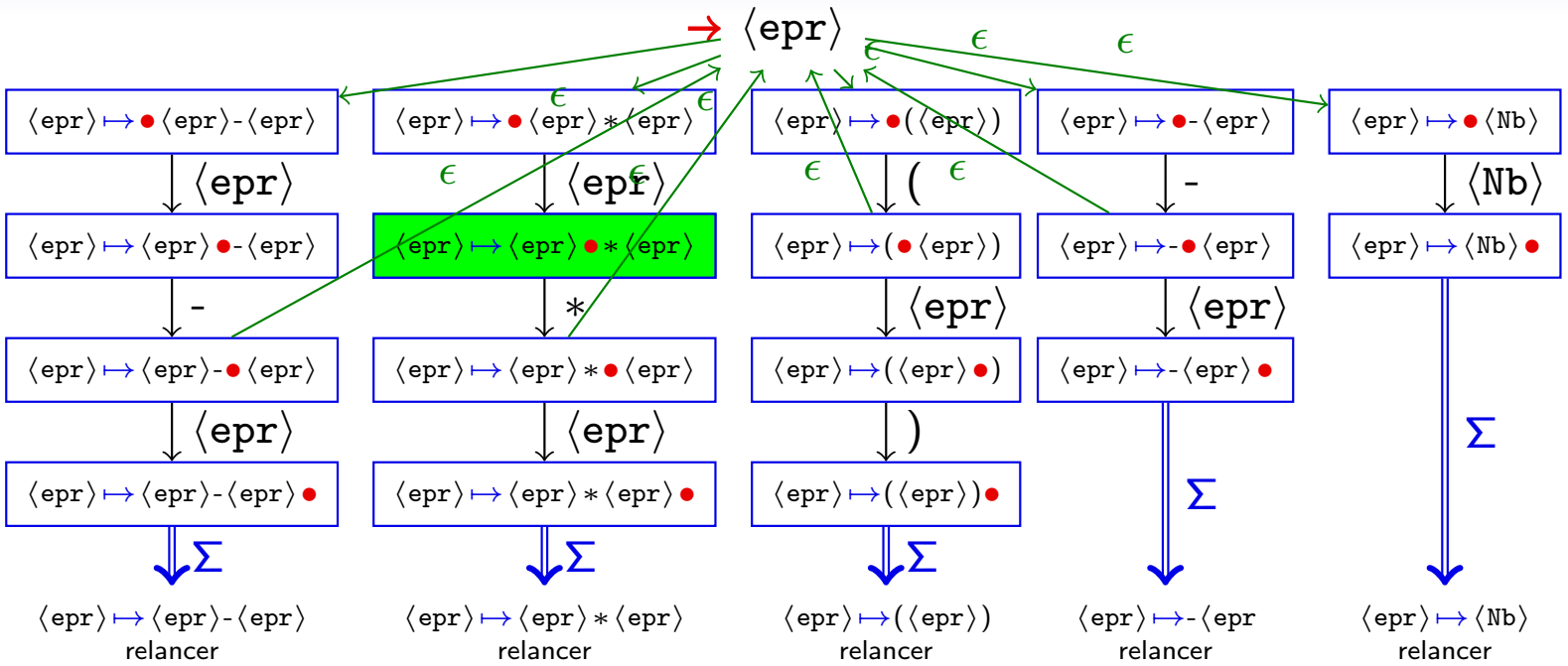


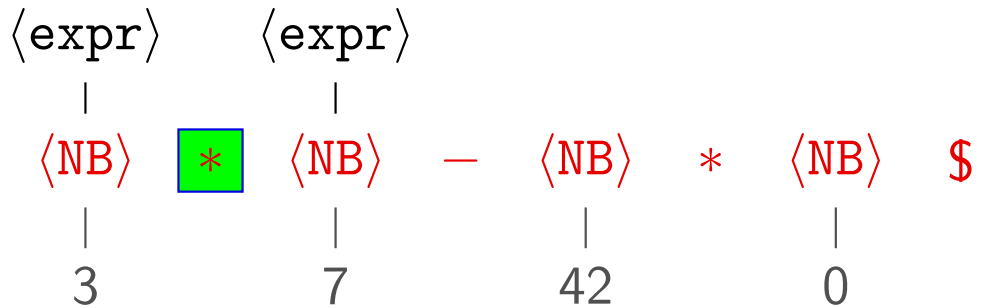
Seconde tentative en action



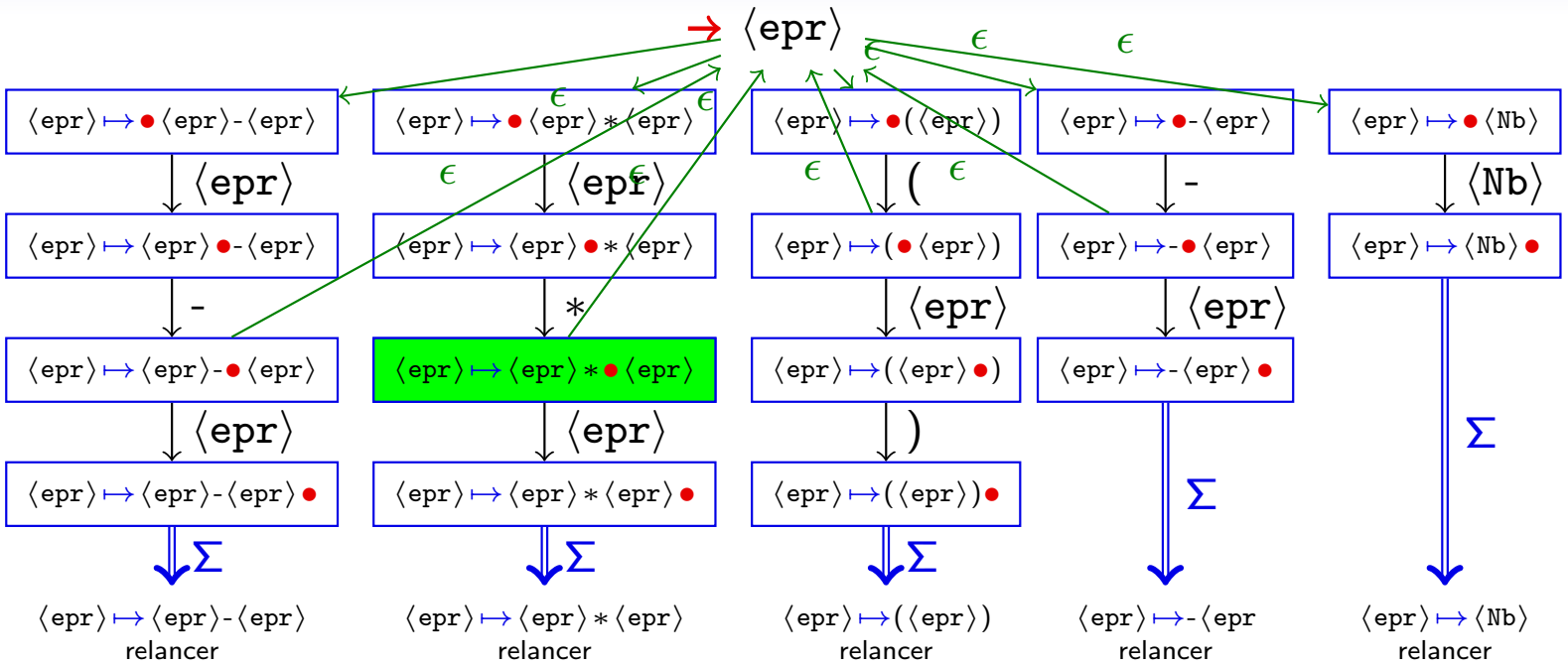


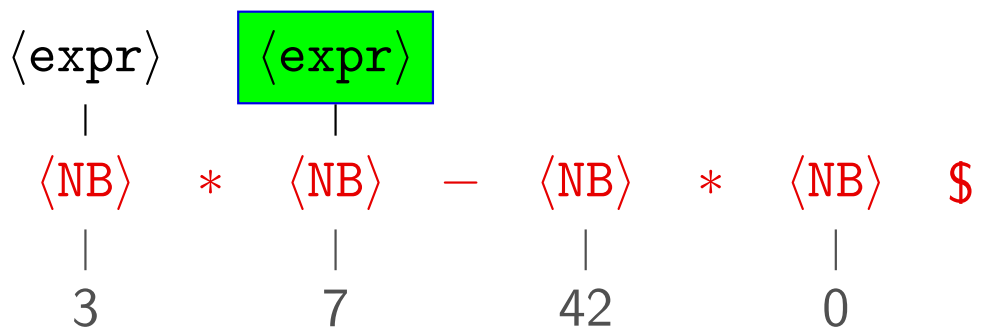
Seconde tentative en action



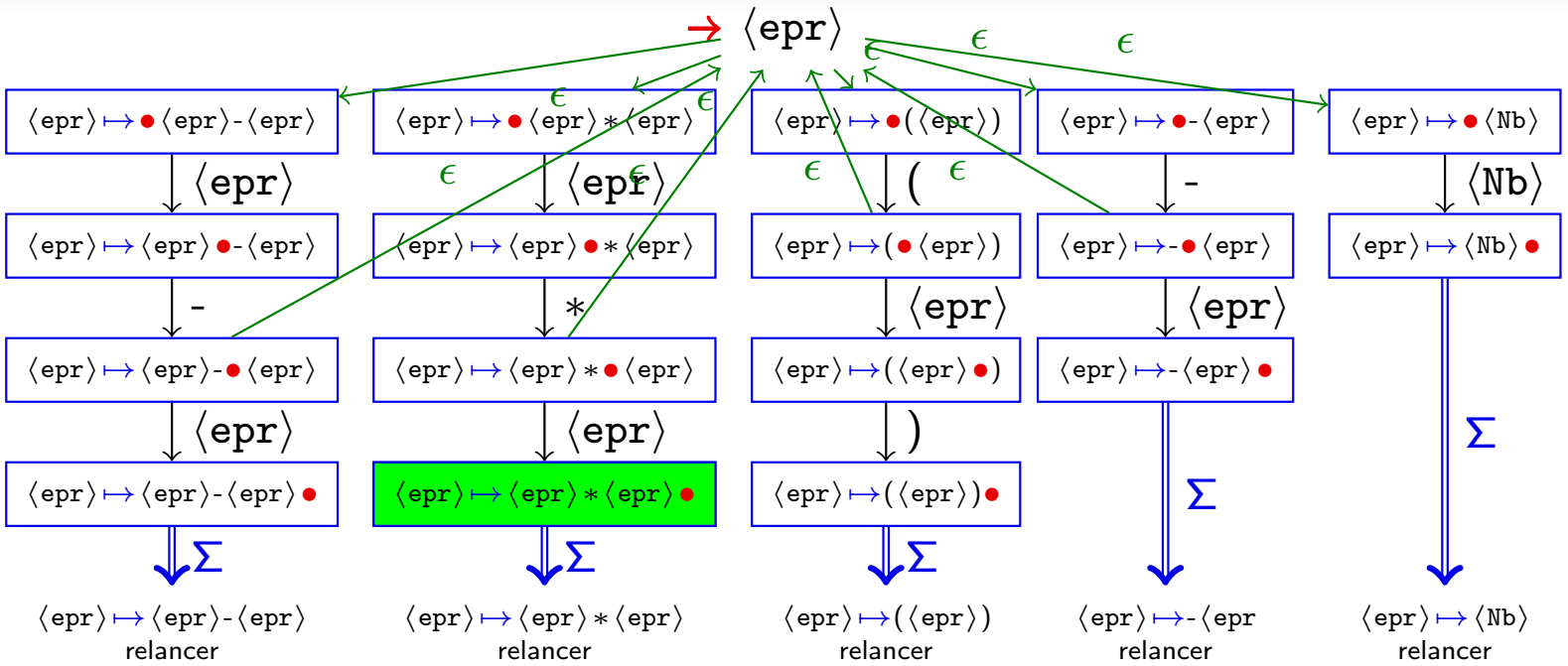


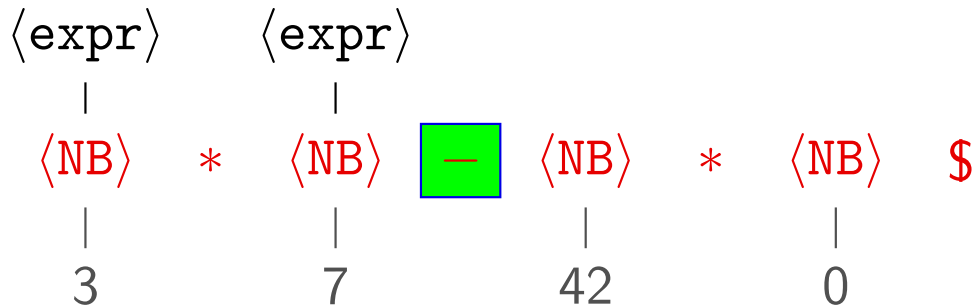
Seconde tentative en action



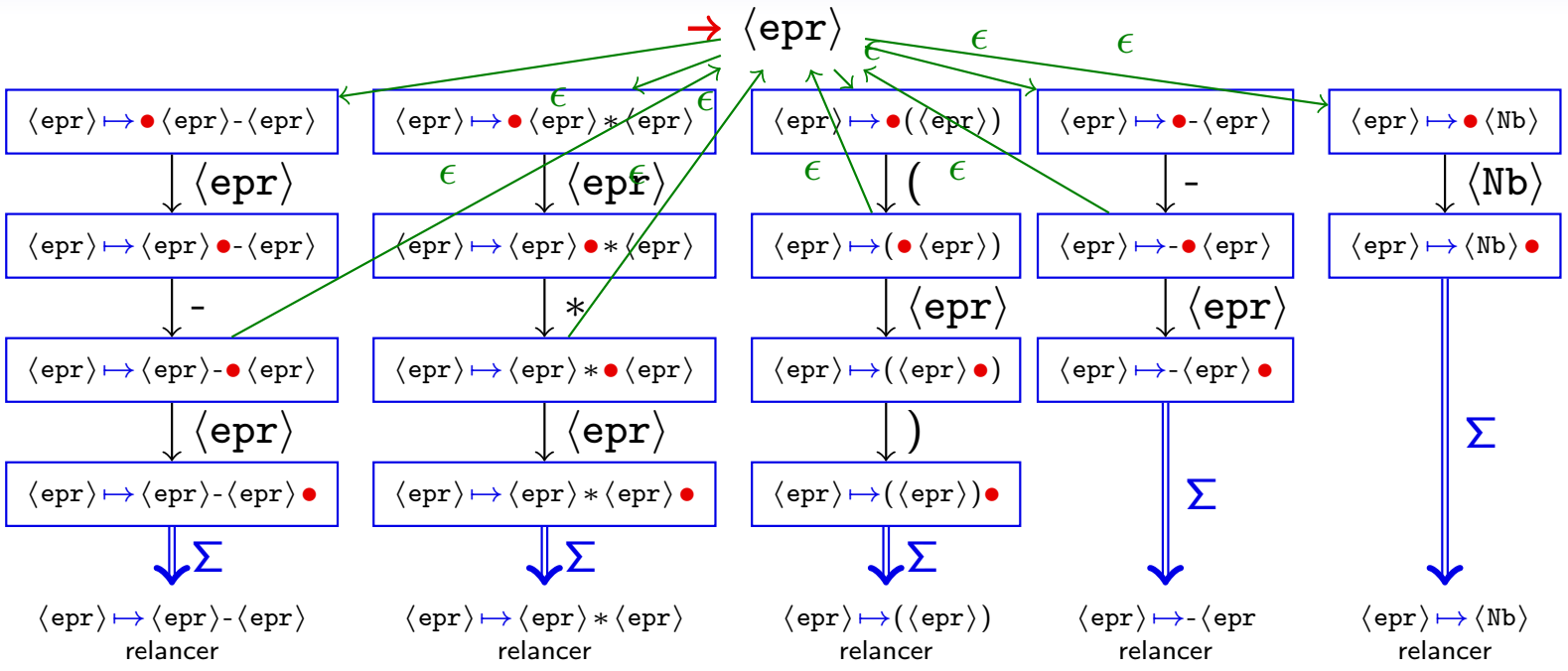


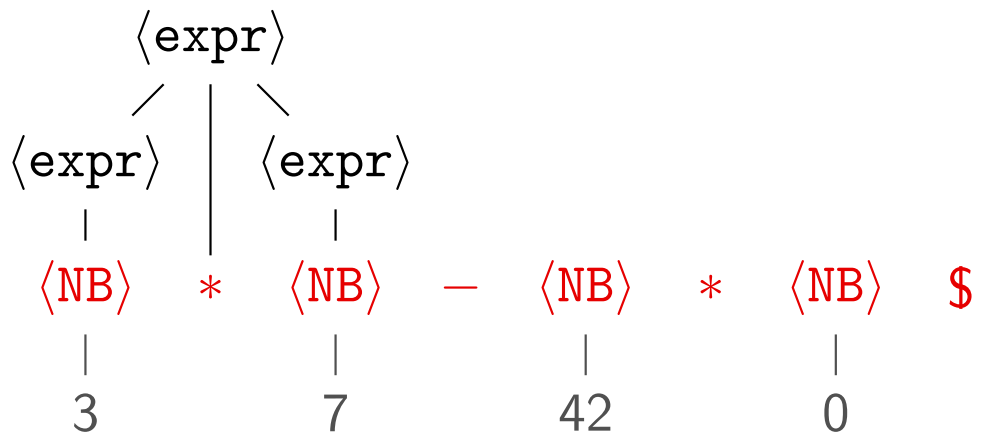
Seconde tentative en action



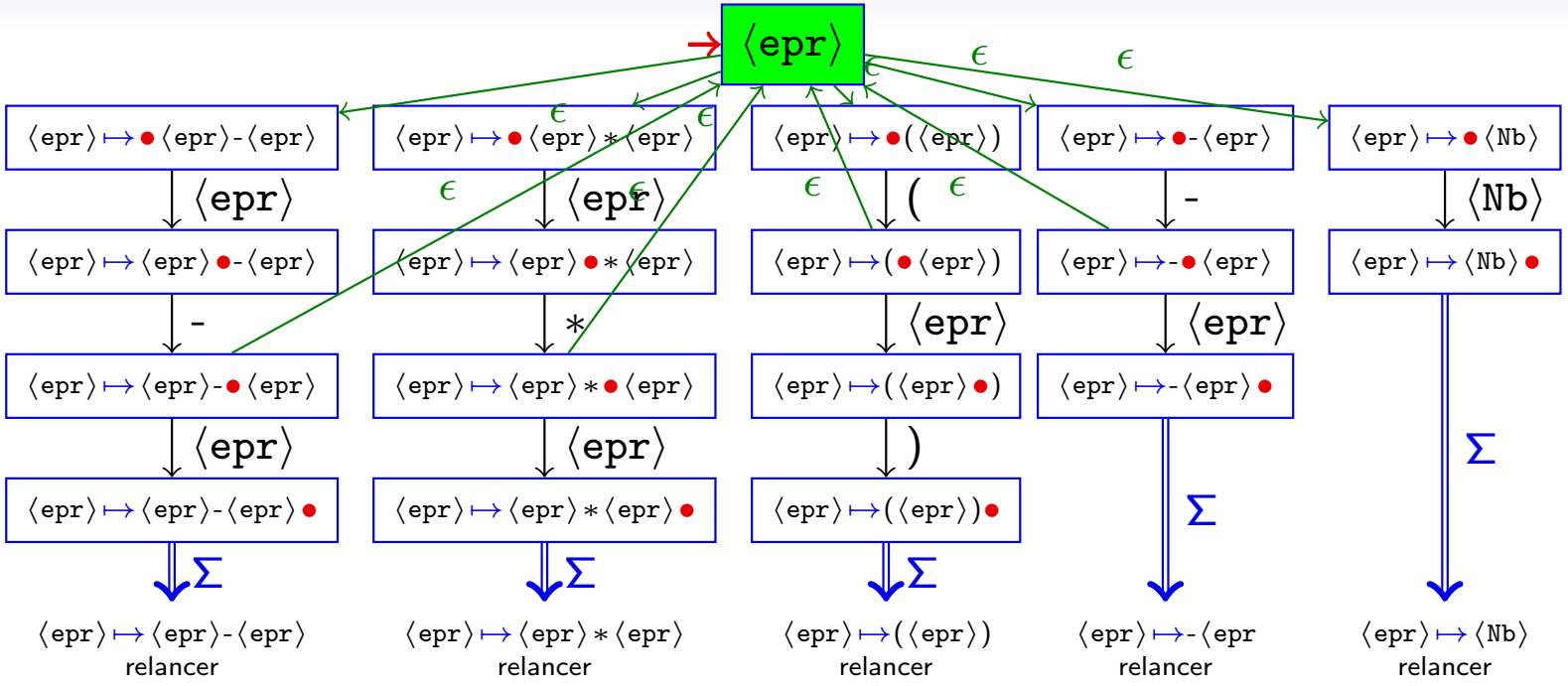


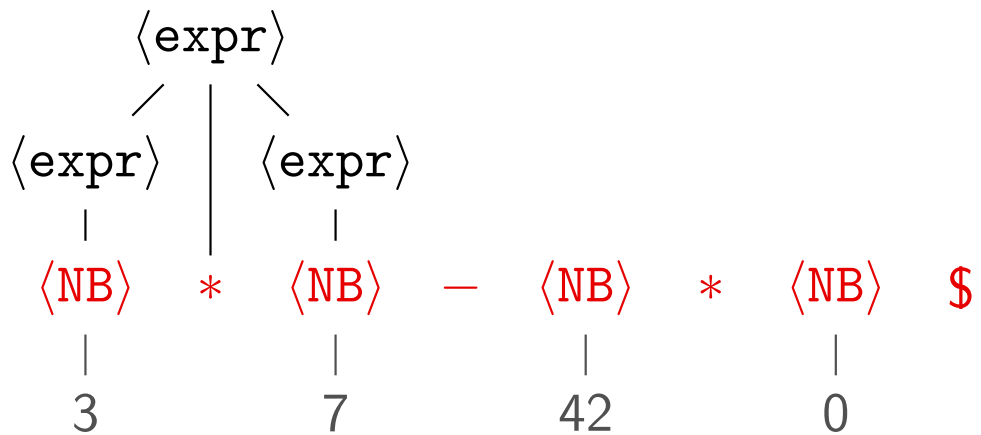
Seconde tentative en action



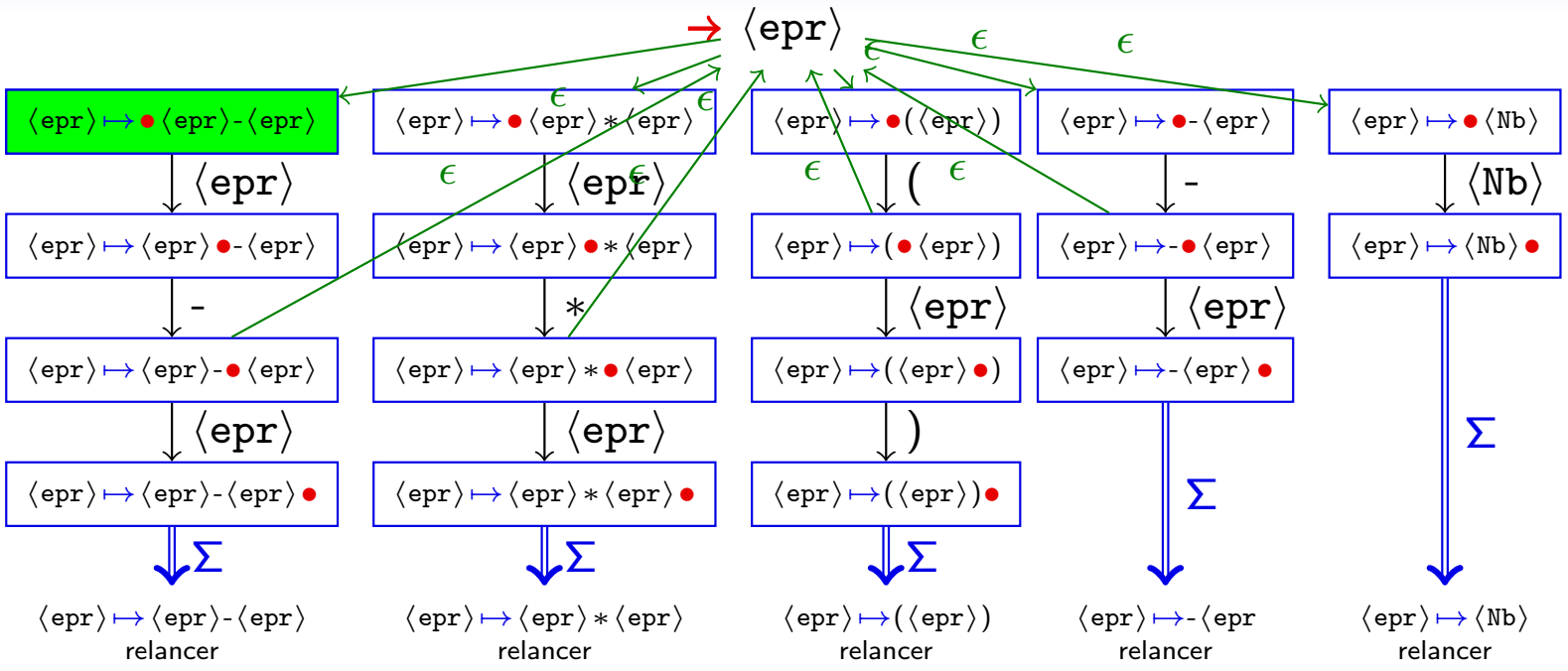


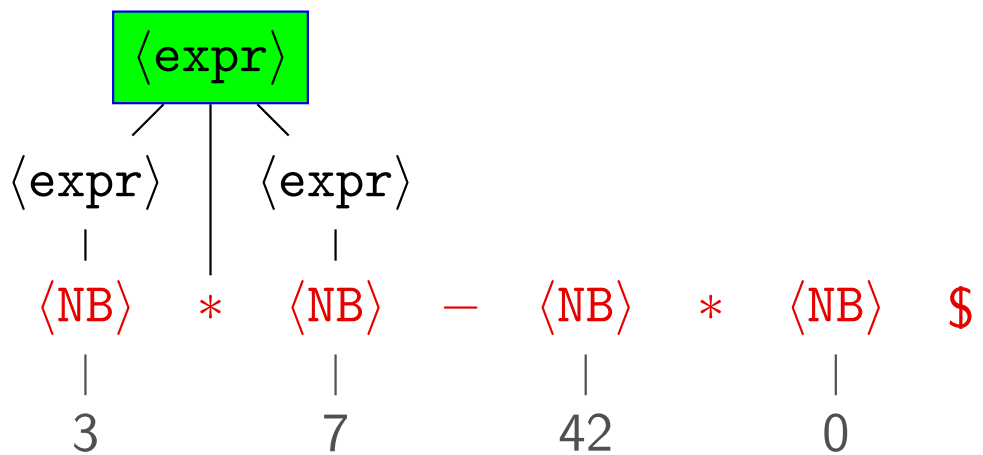
Seconde tentative en action



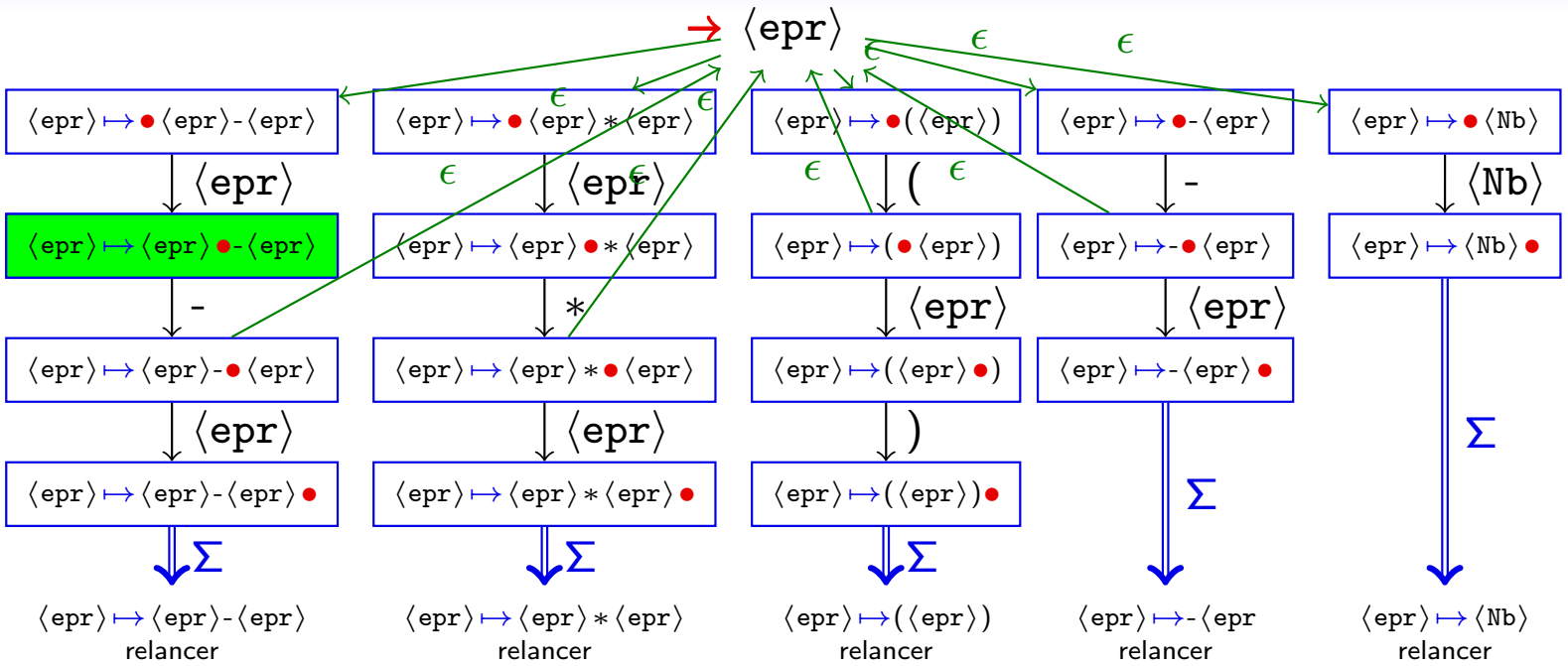


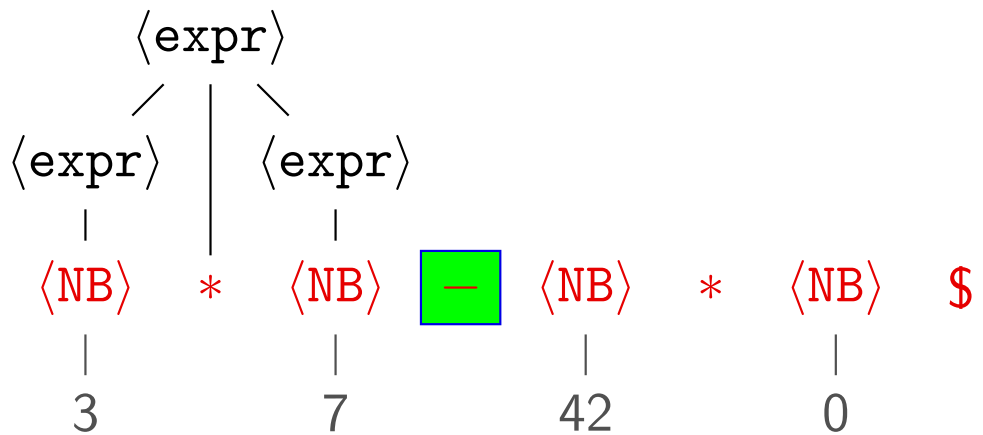
Seconde tentative en action



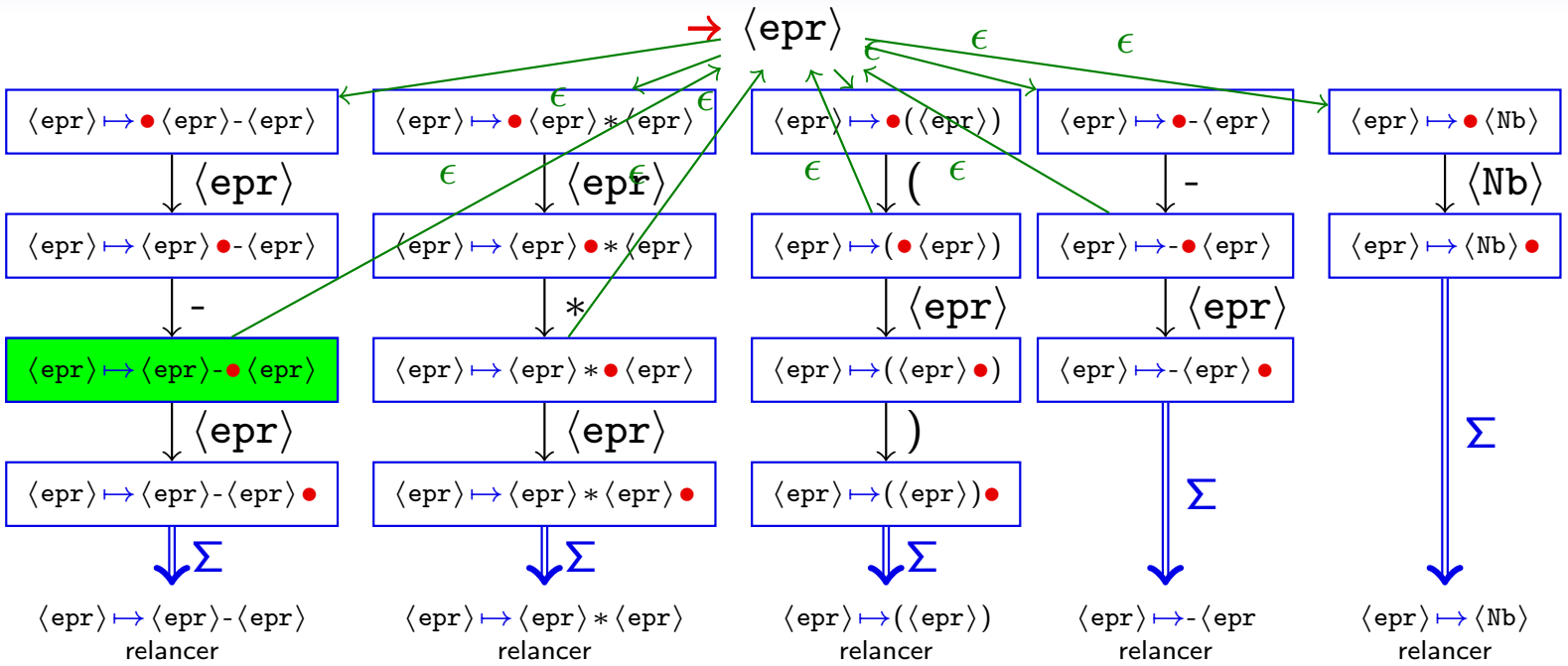


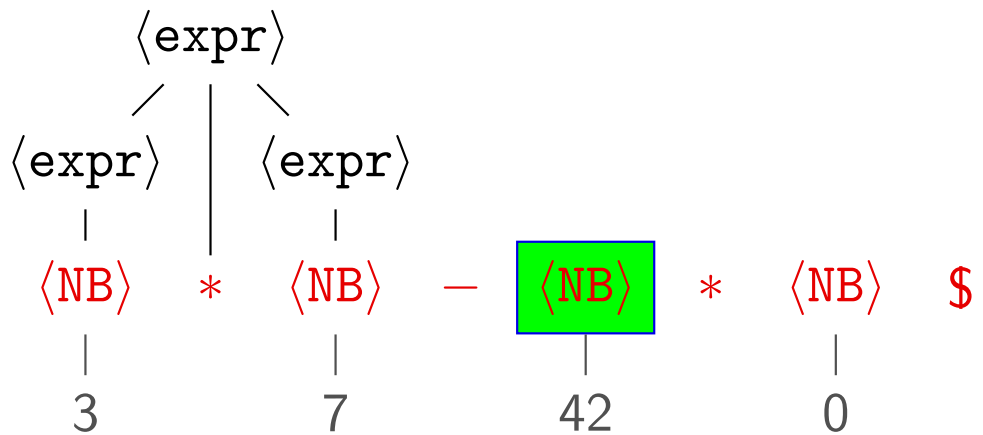
Seconde tentative en action



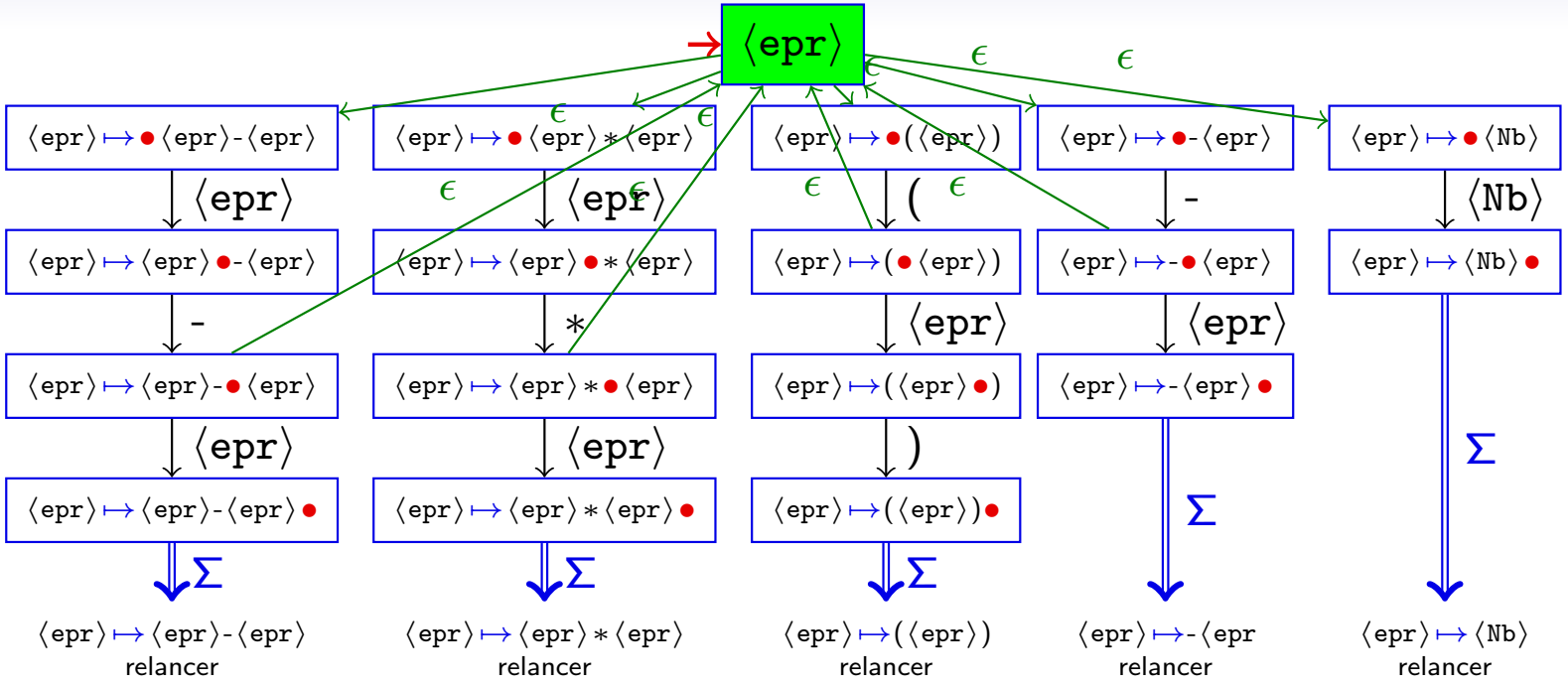


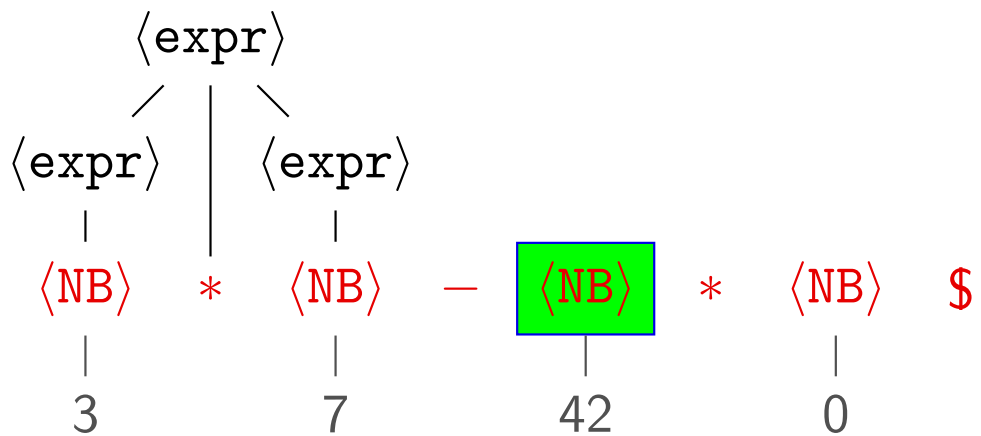
Seconde tentative en action



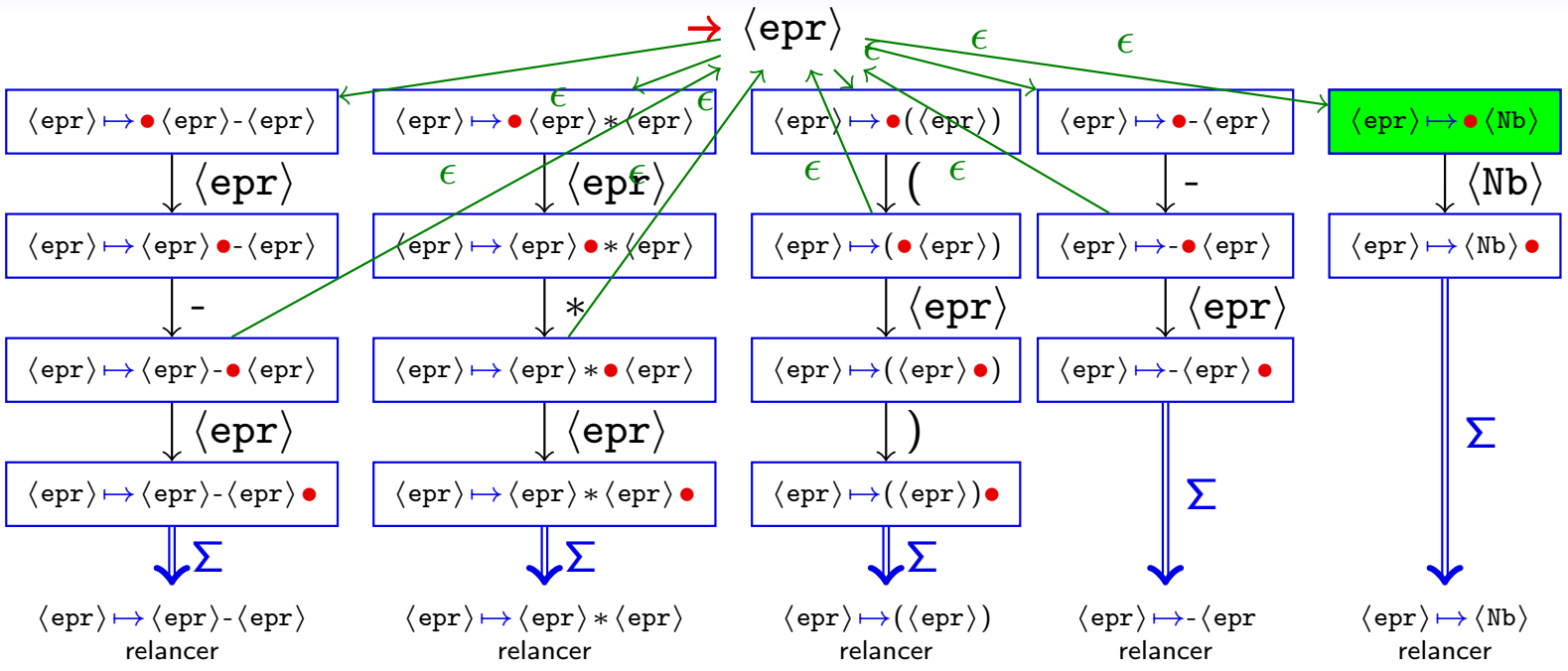


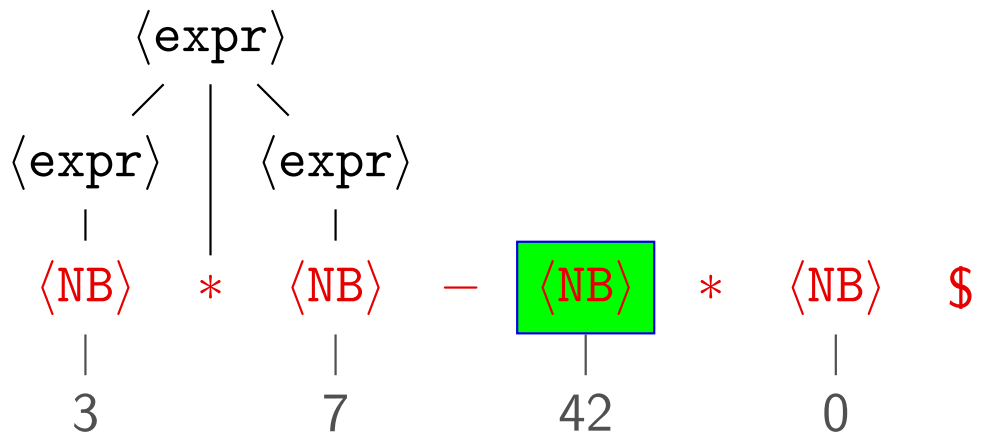
Seconde tentative en action



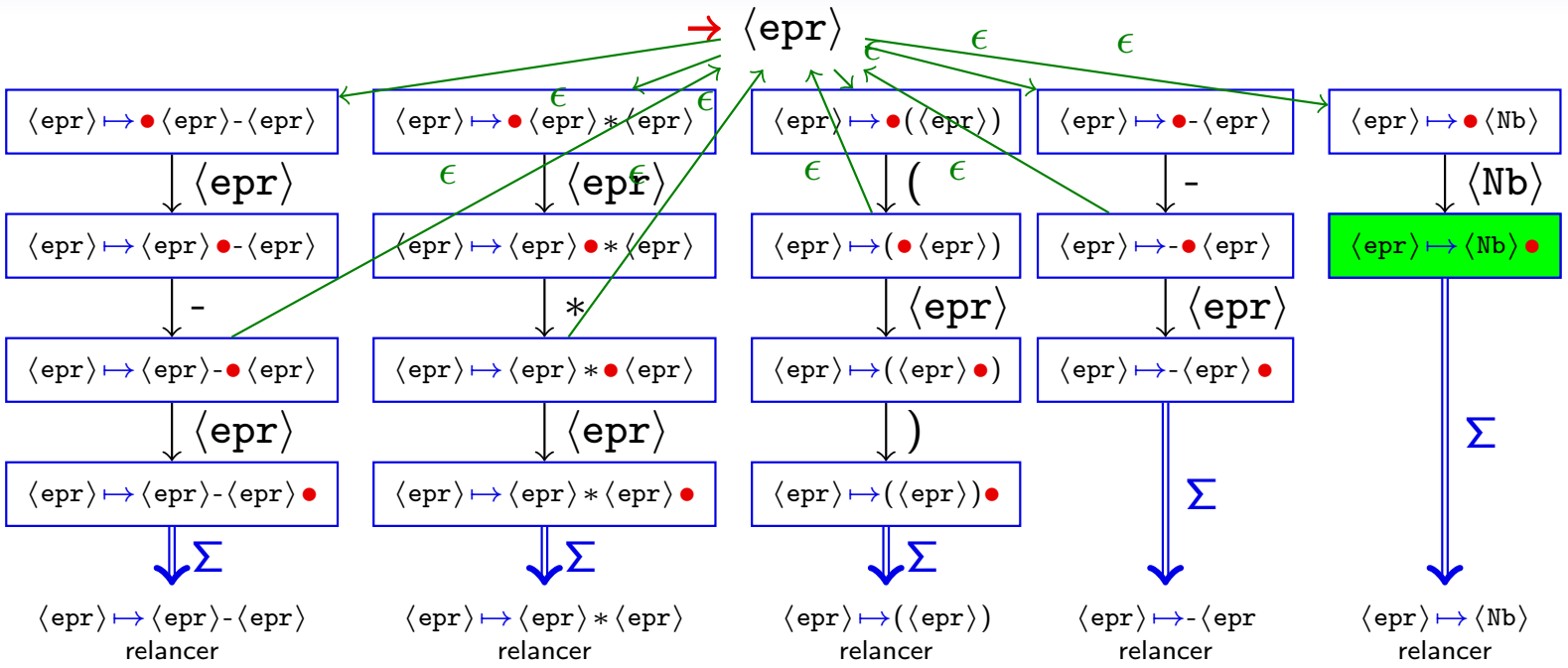


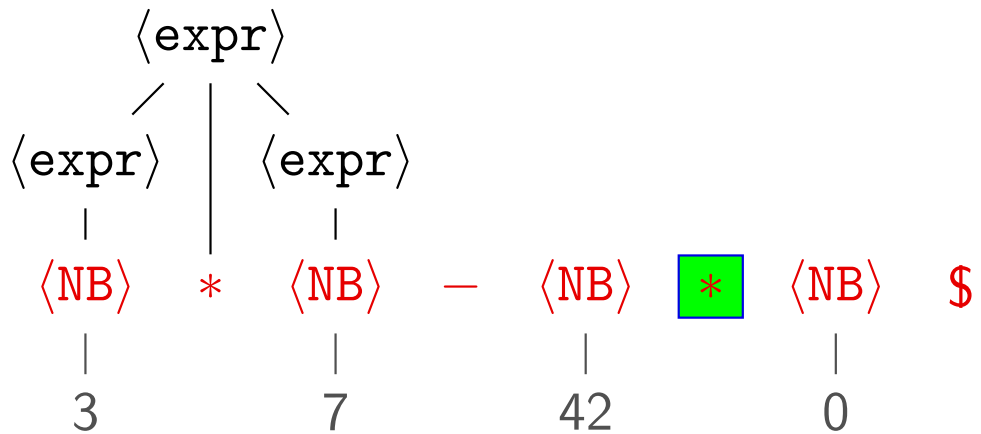
Seconde tentative en action



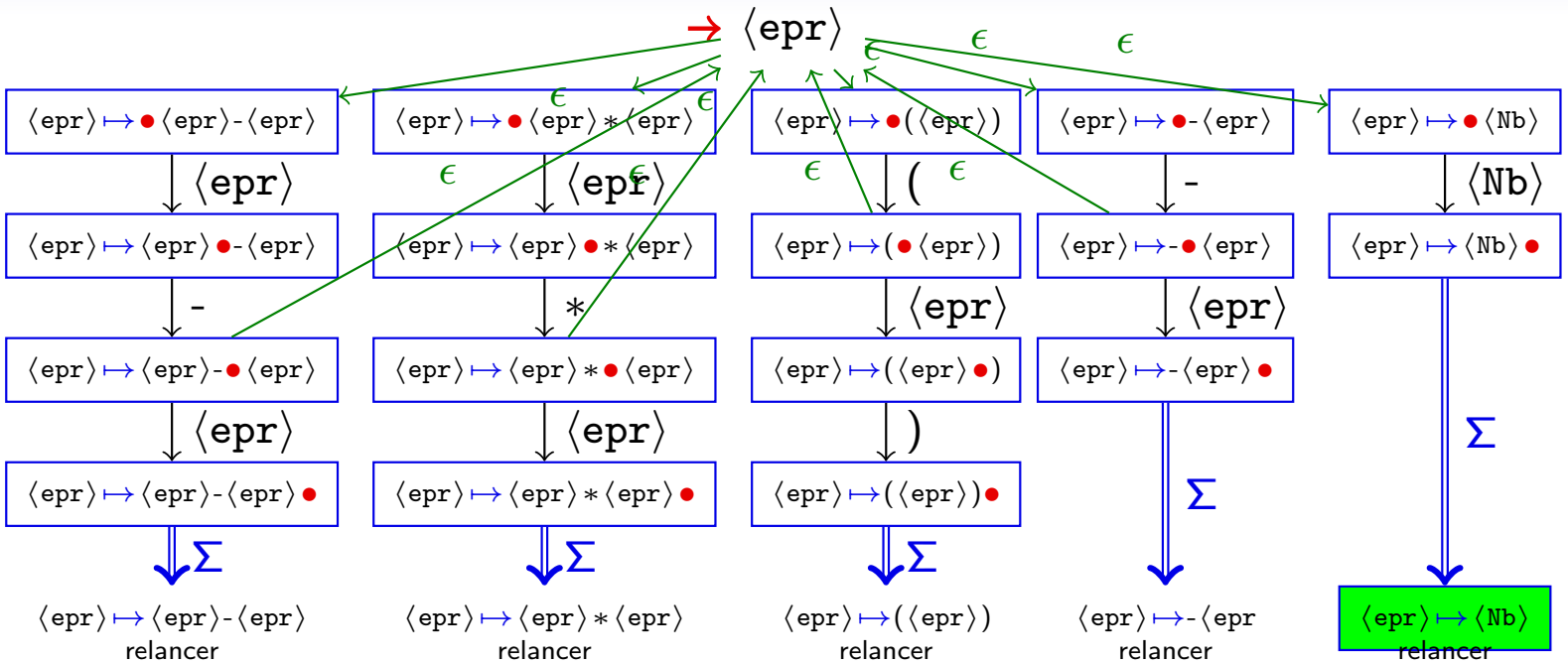


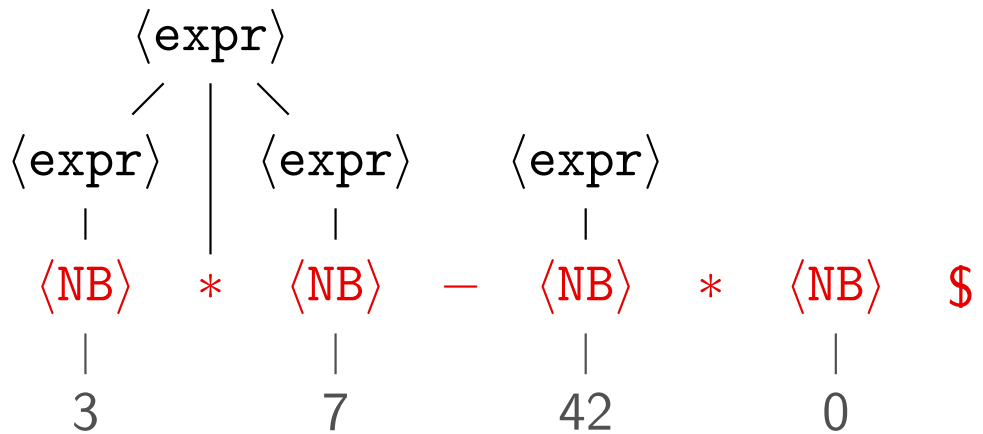
Seconde tentative en action



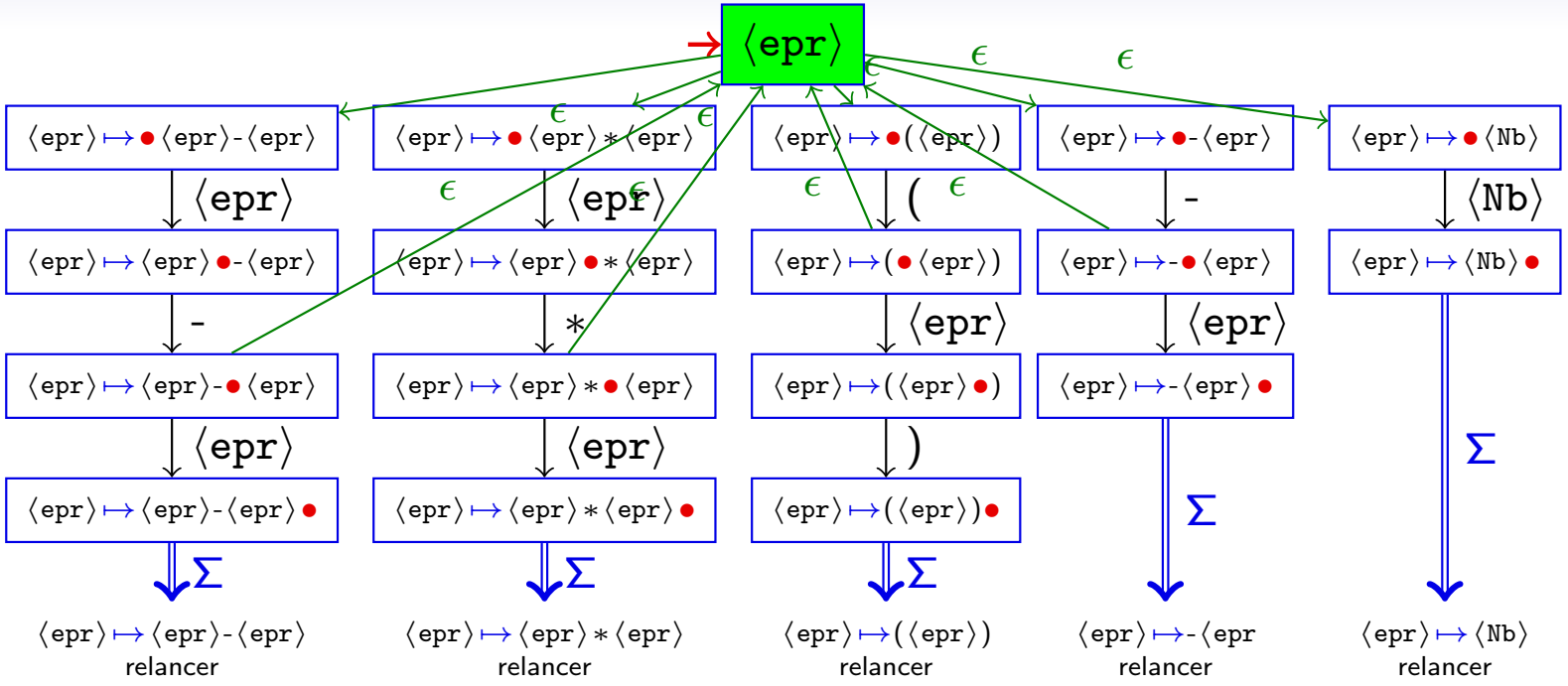


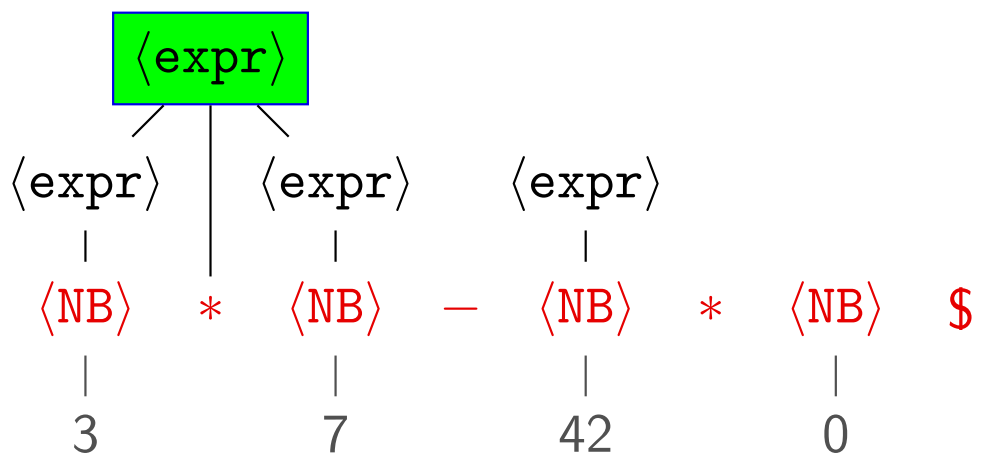
Seconde tentative en action



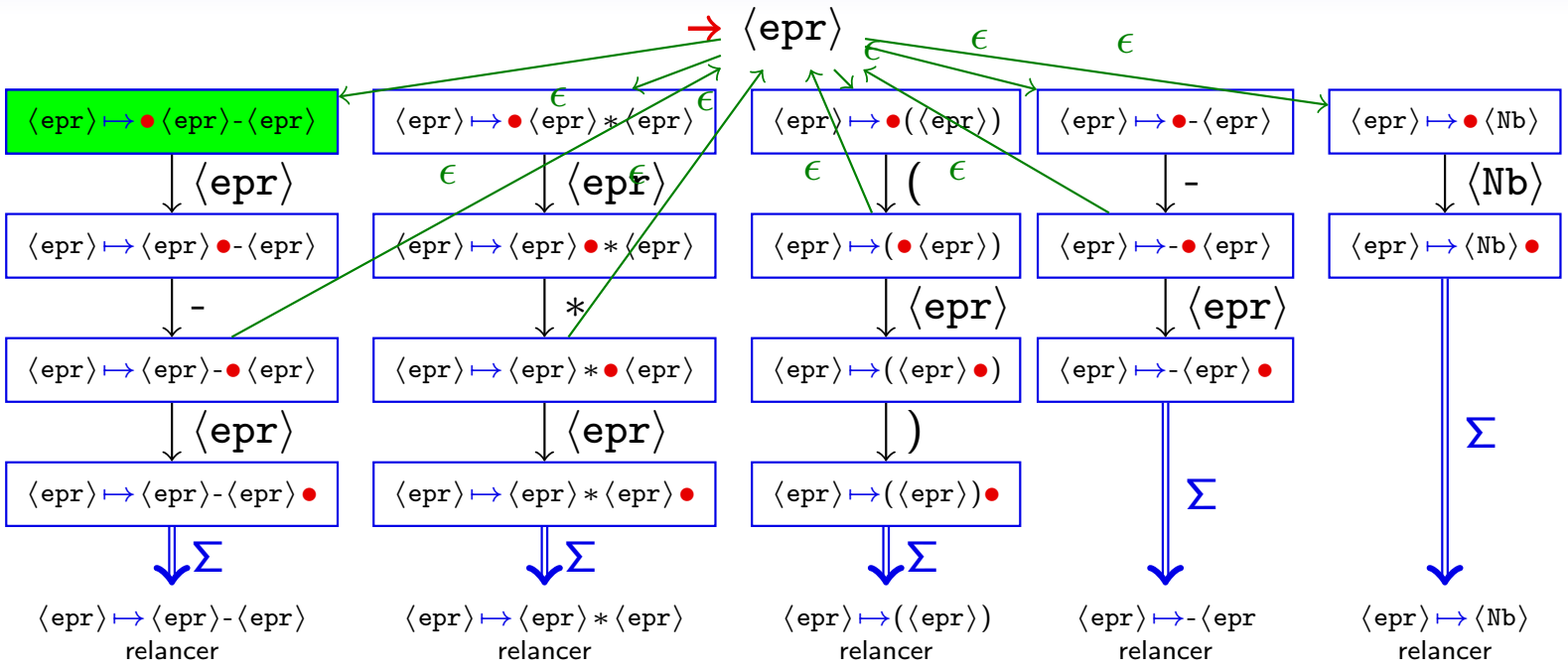


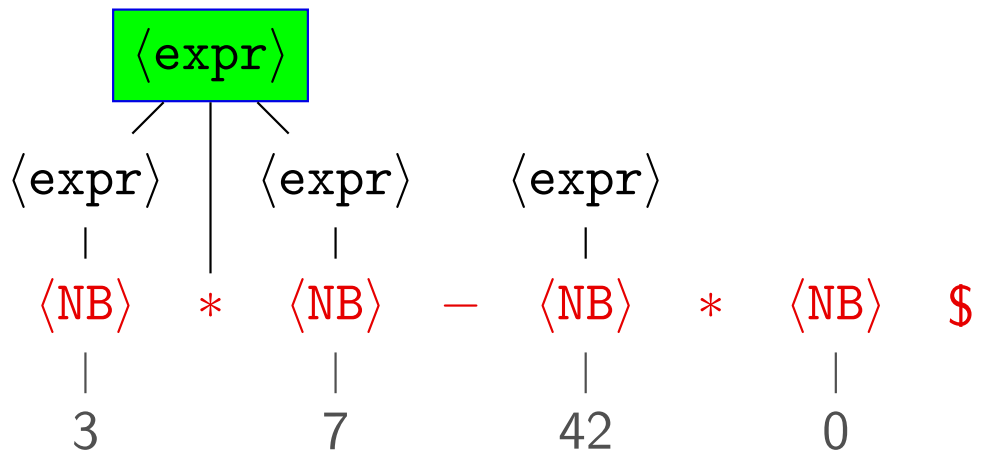
Seconde tentative en action



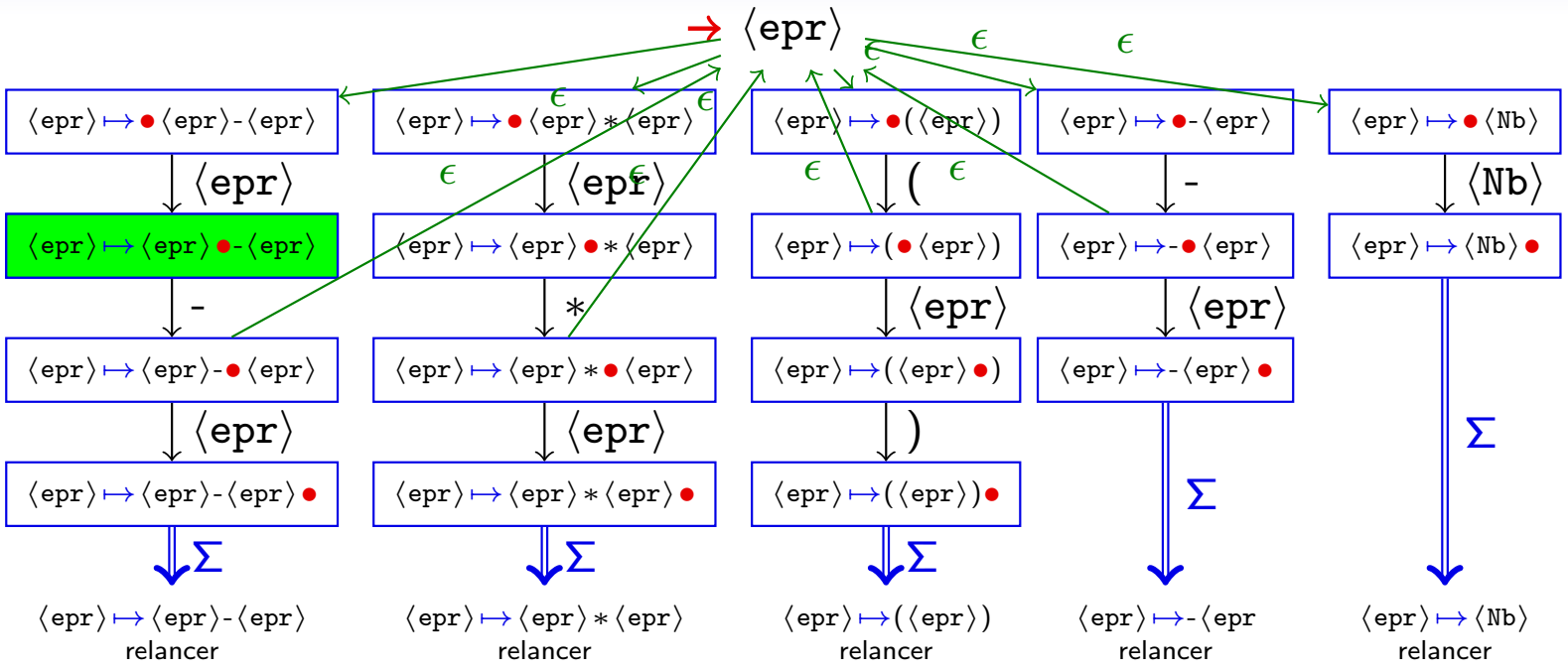


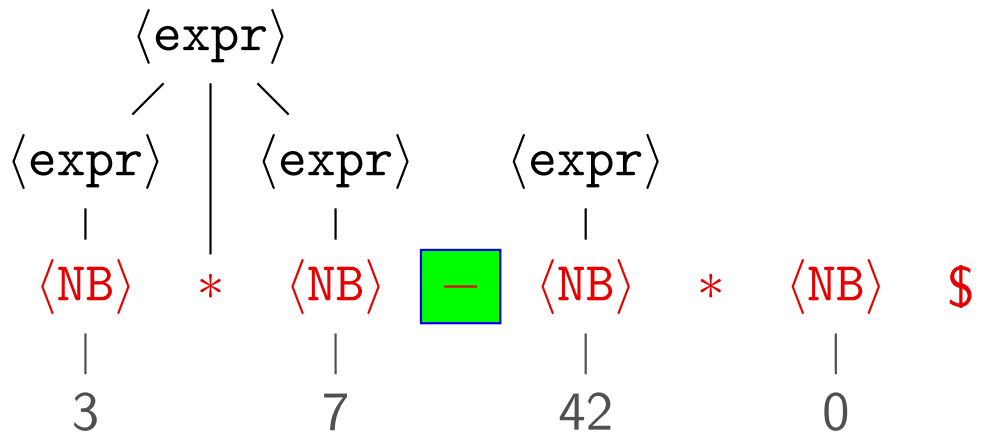
Seconde tentative en action



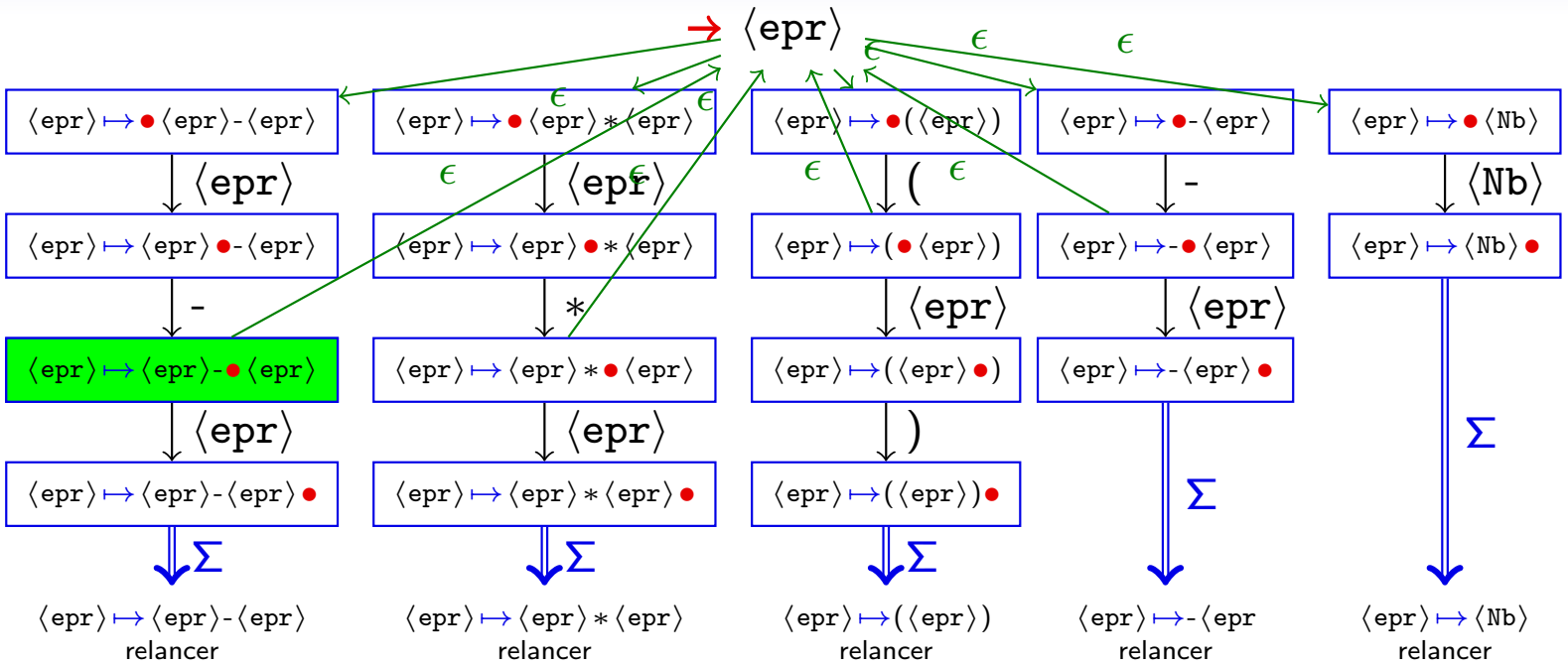


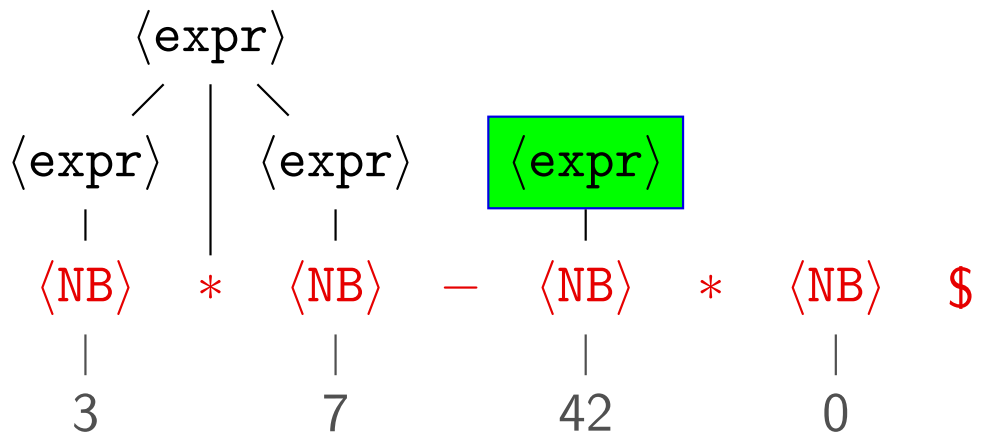
Seconde tentative en action



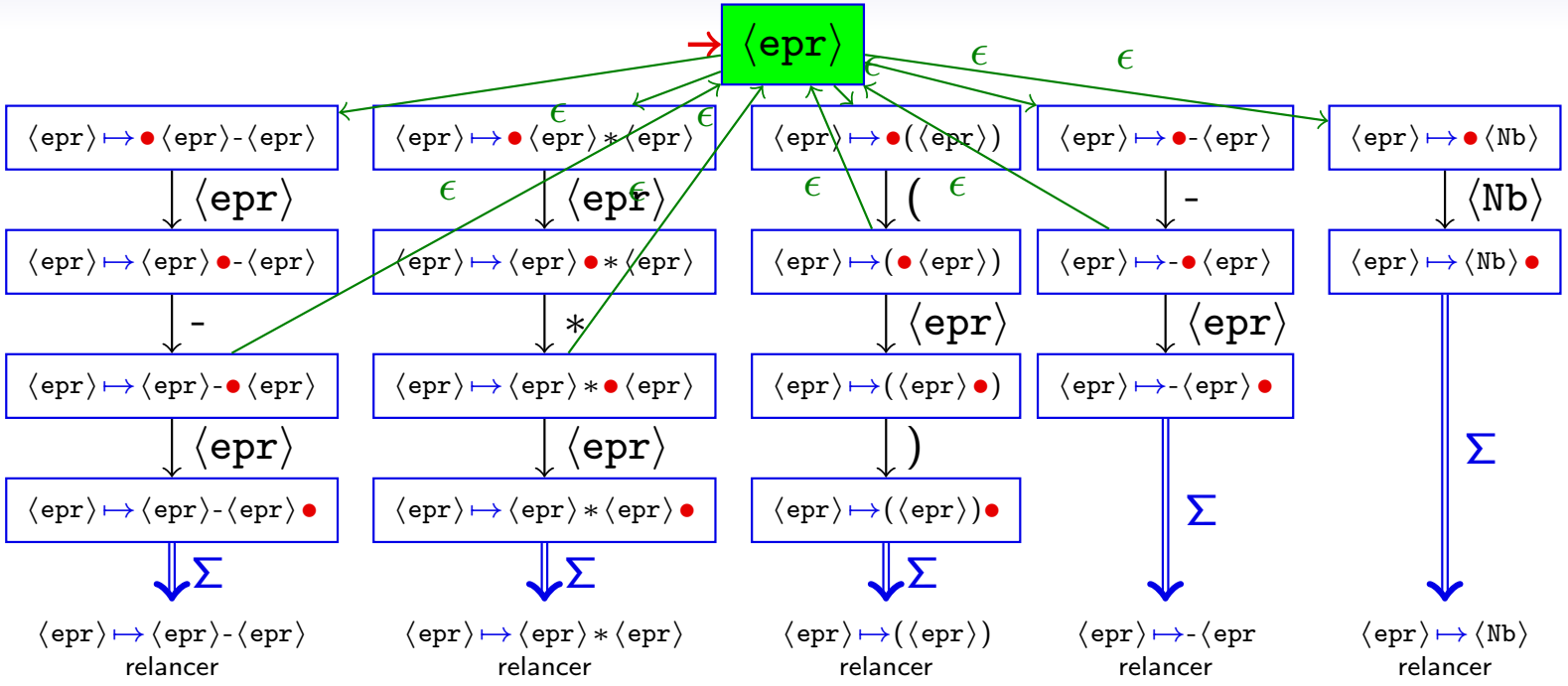


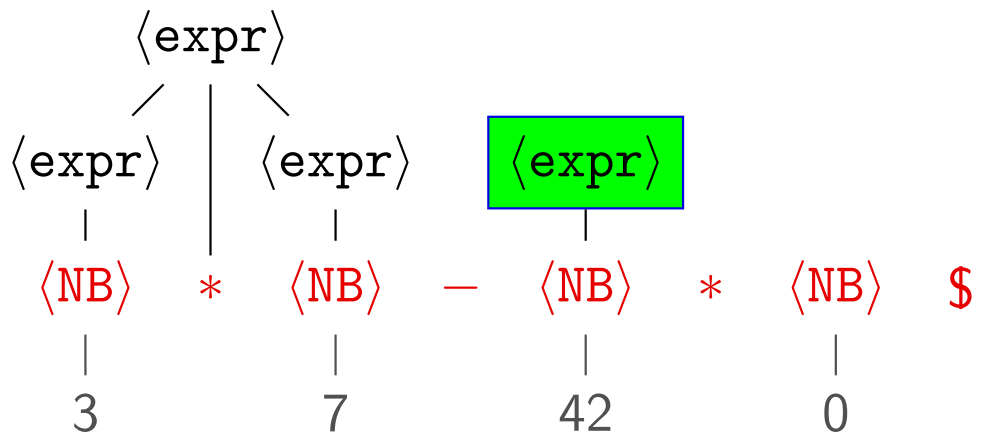
Seconde tentative en action



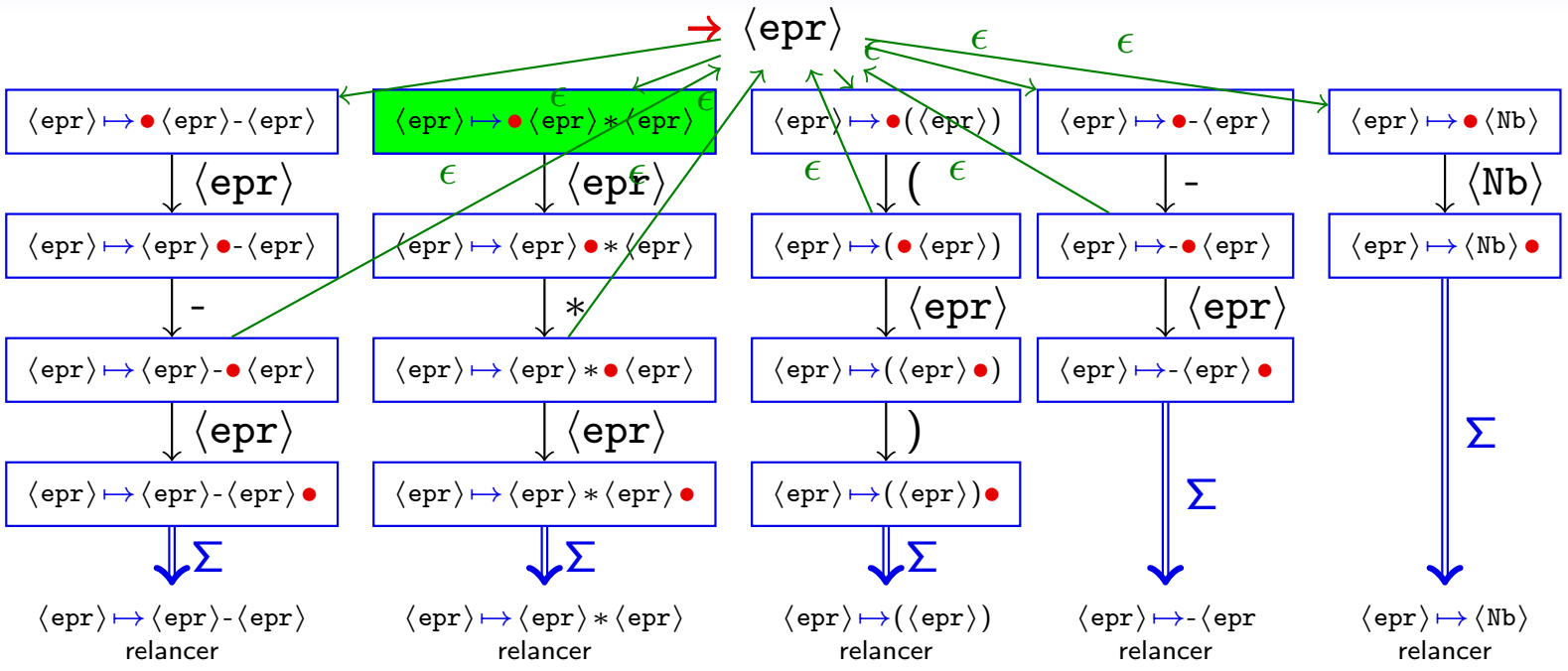


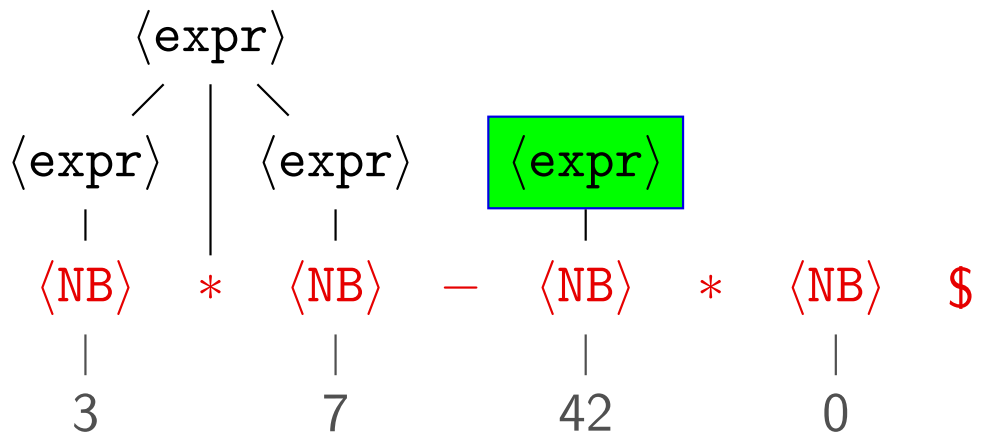
Seconde tentative en action



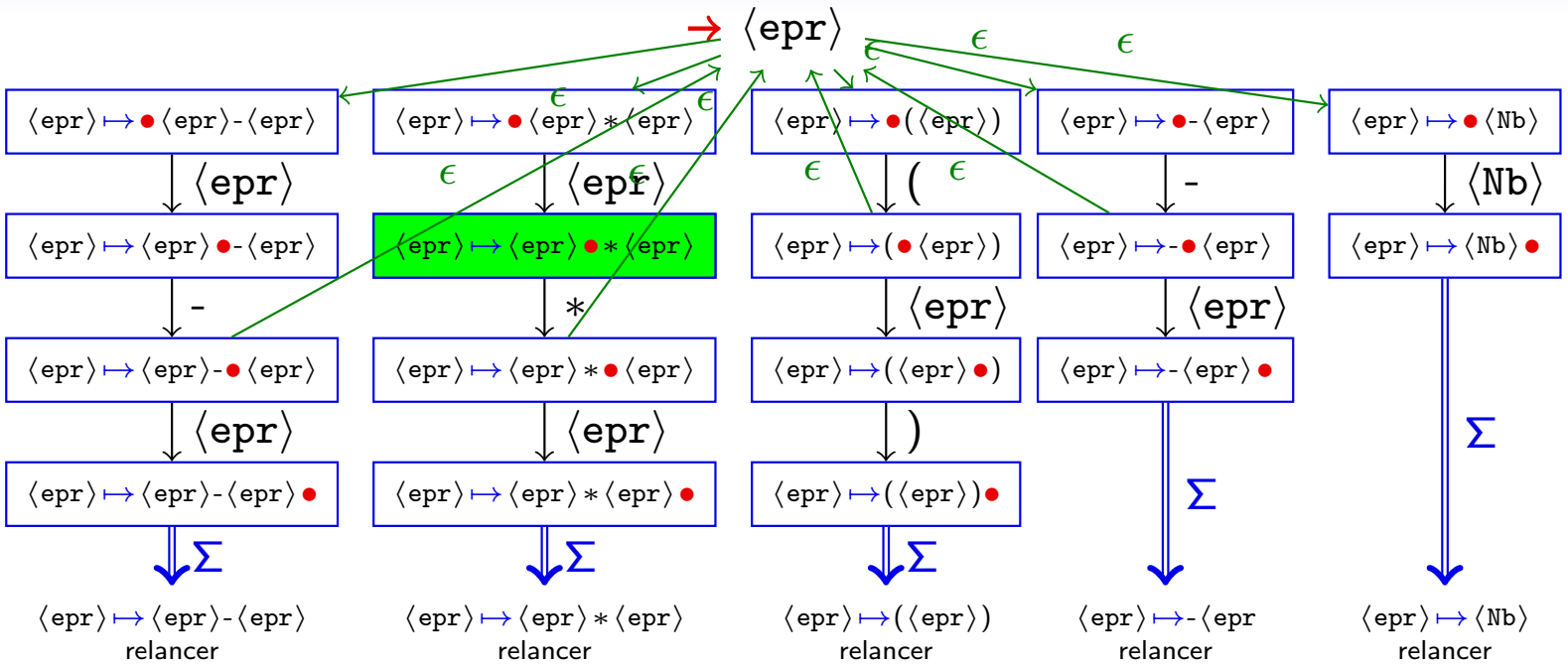


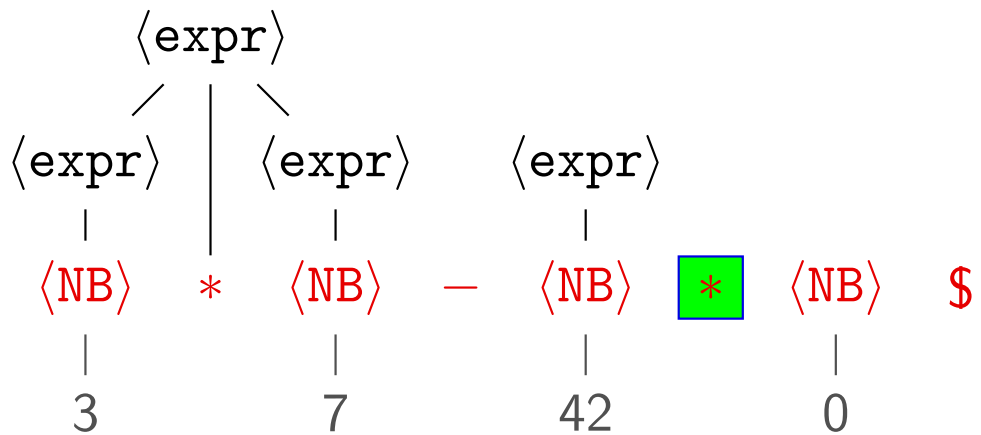
Seconde tentative en action



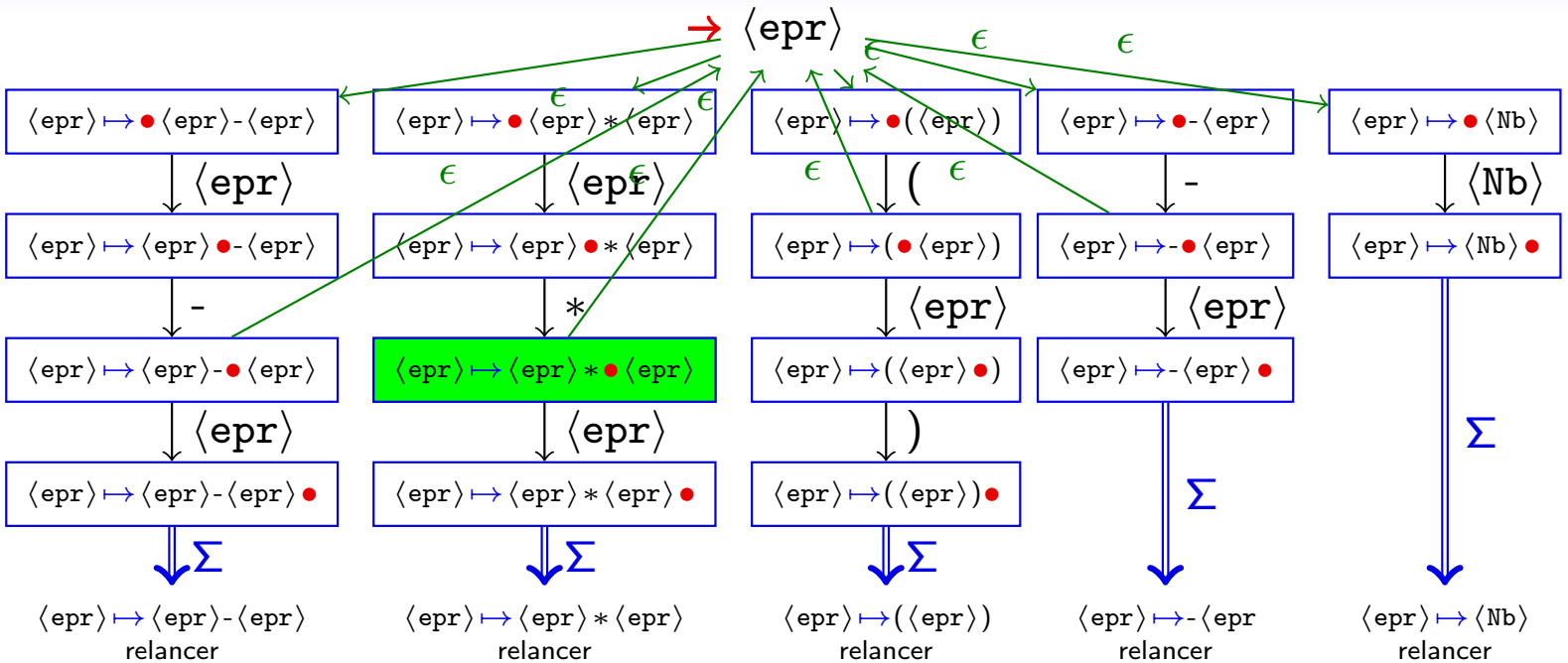


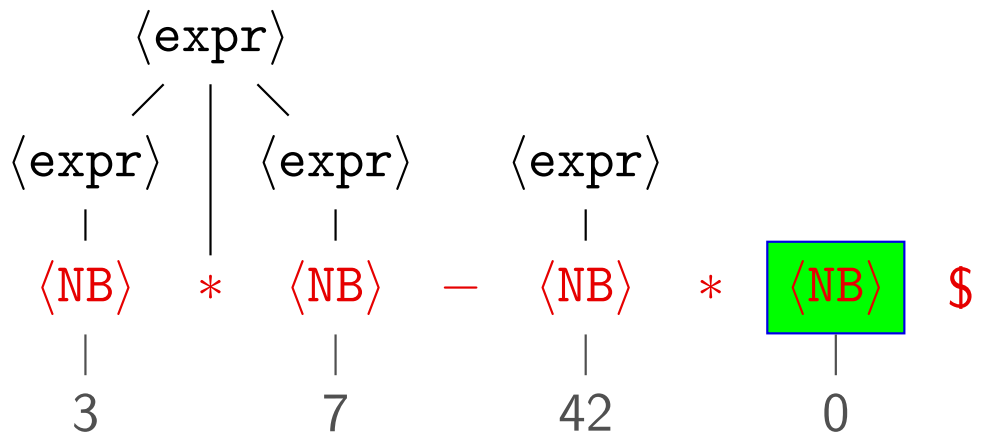
Seconde tentative en action



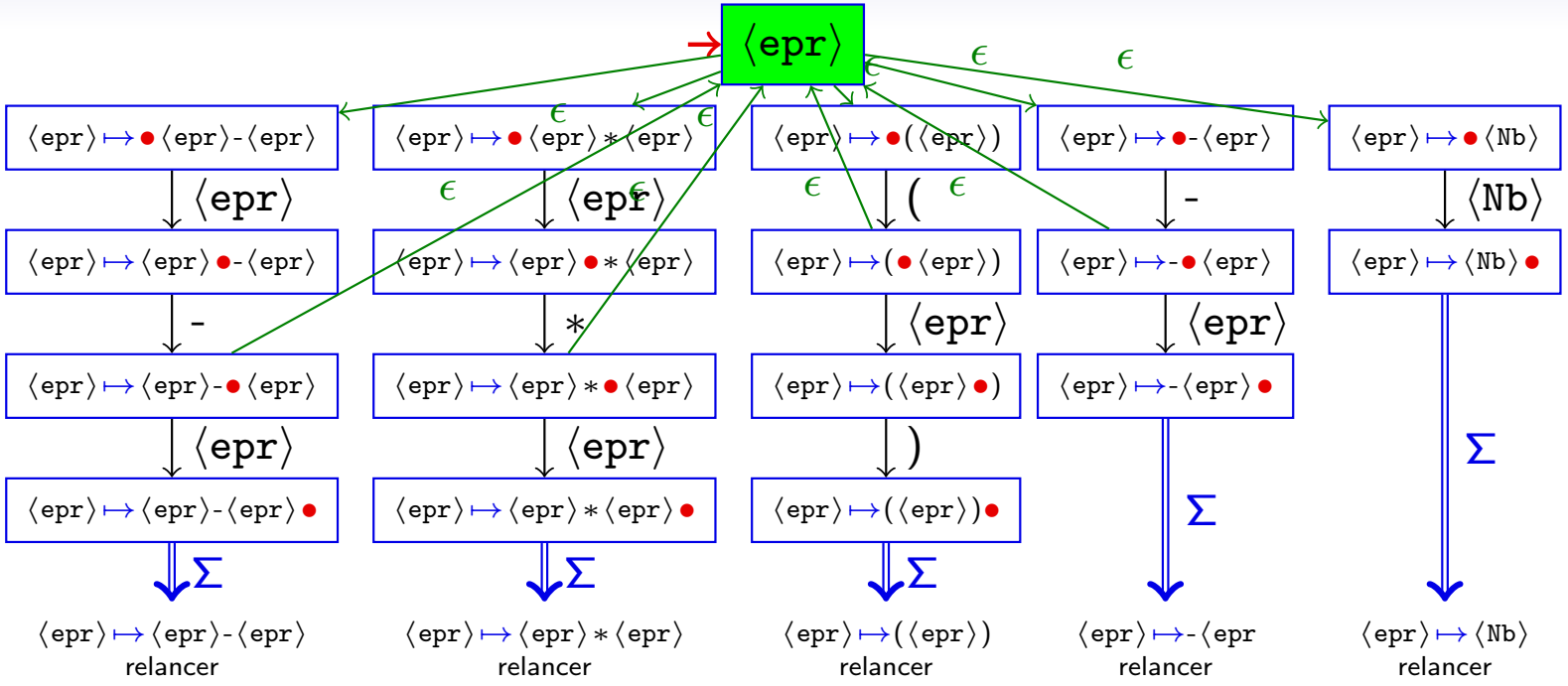


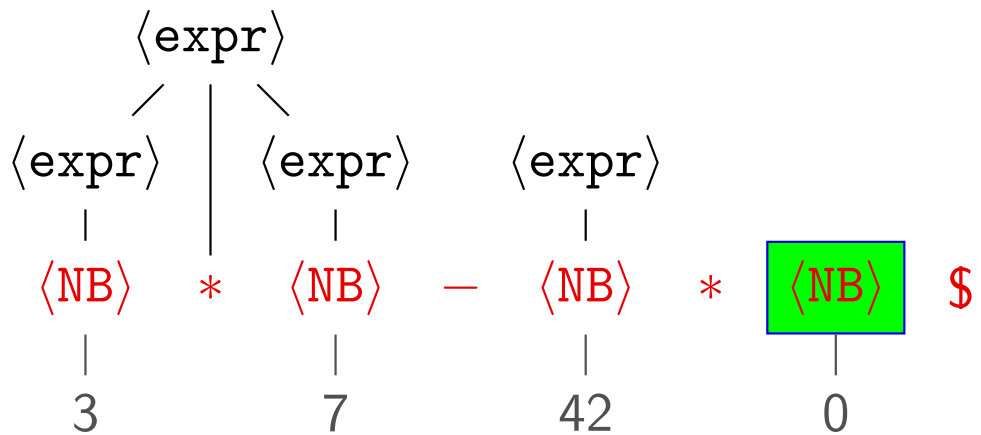
Seconde tentative en action



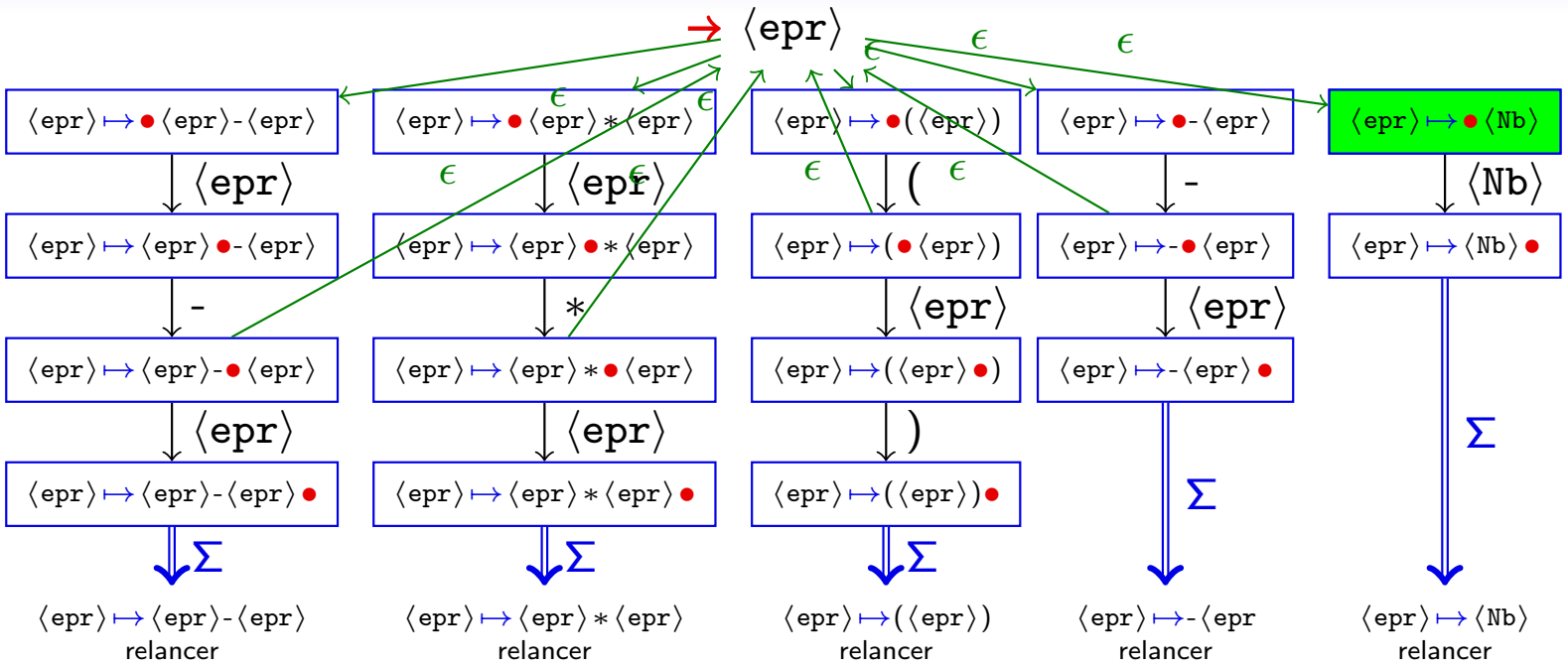


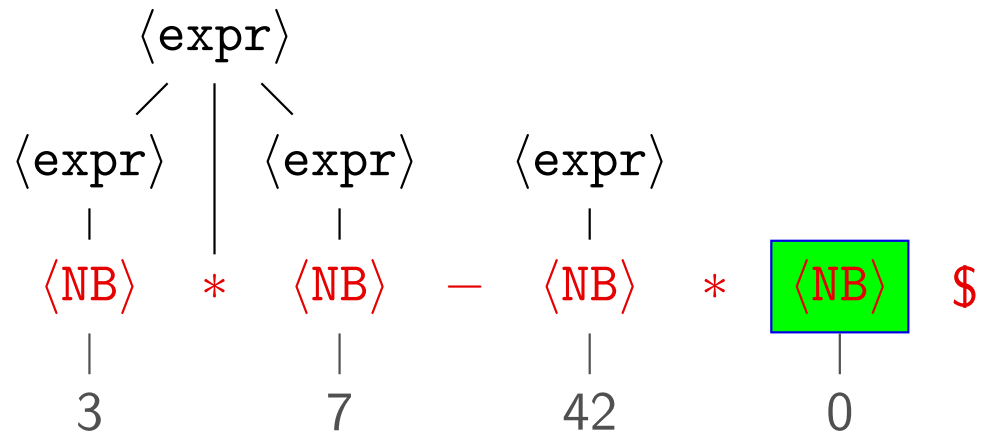
Seconde tentative en action



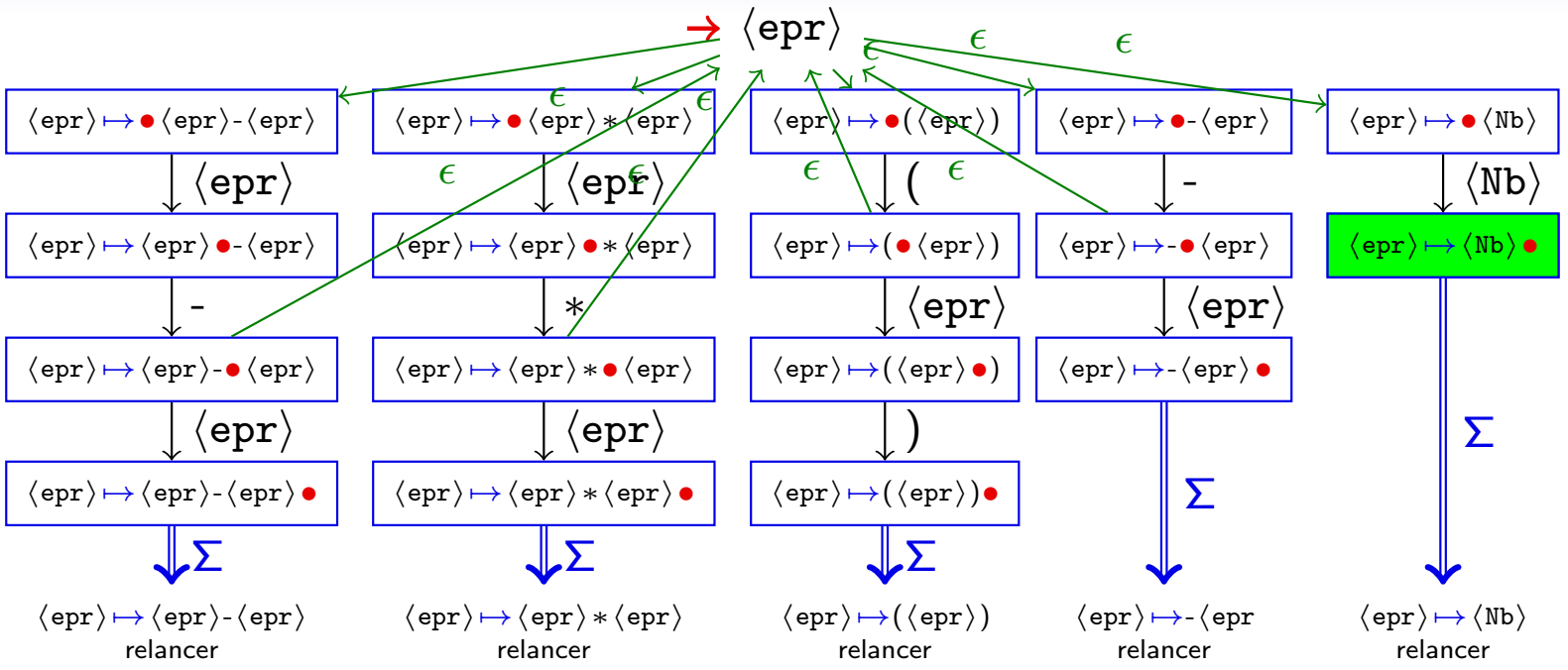


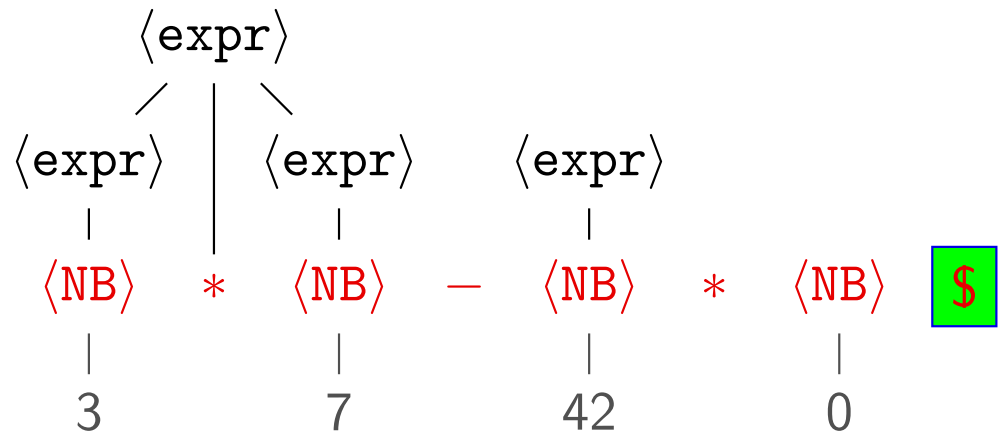
Seconde tentative en action



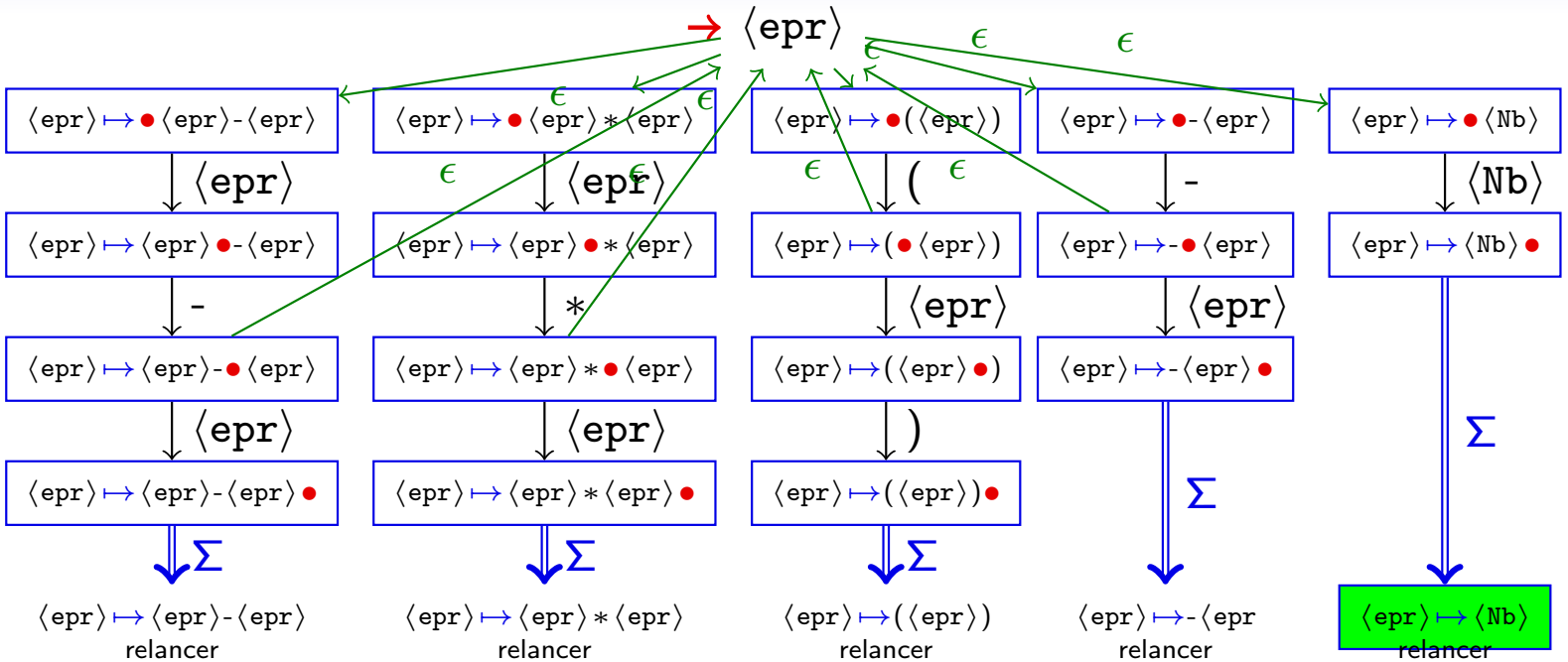


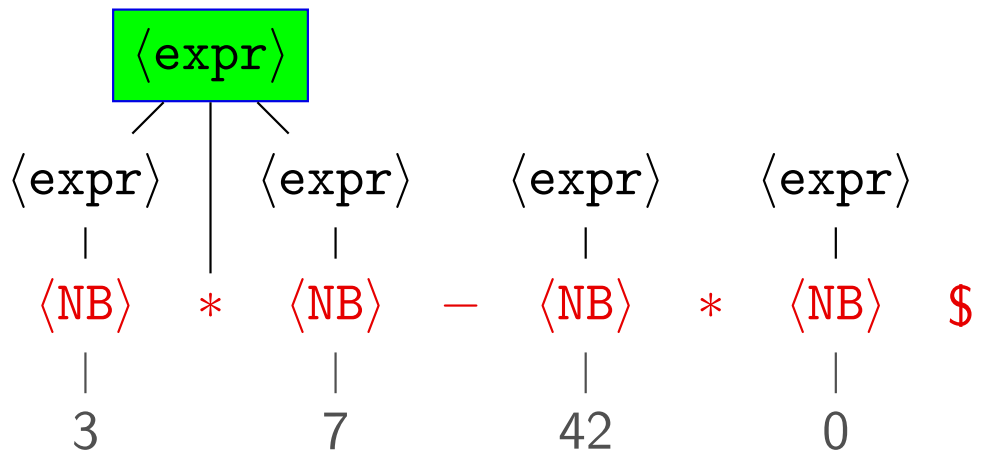
Seconde tentative en action



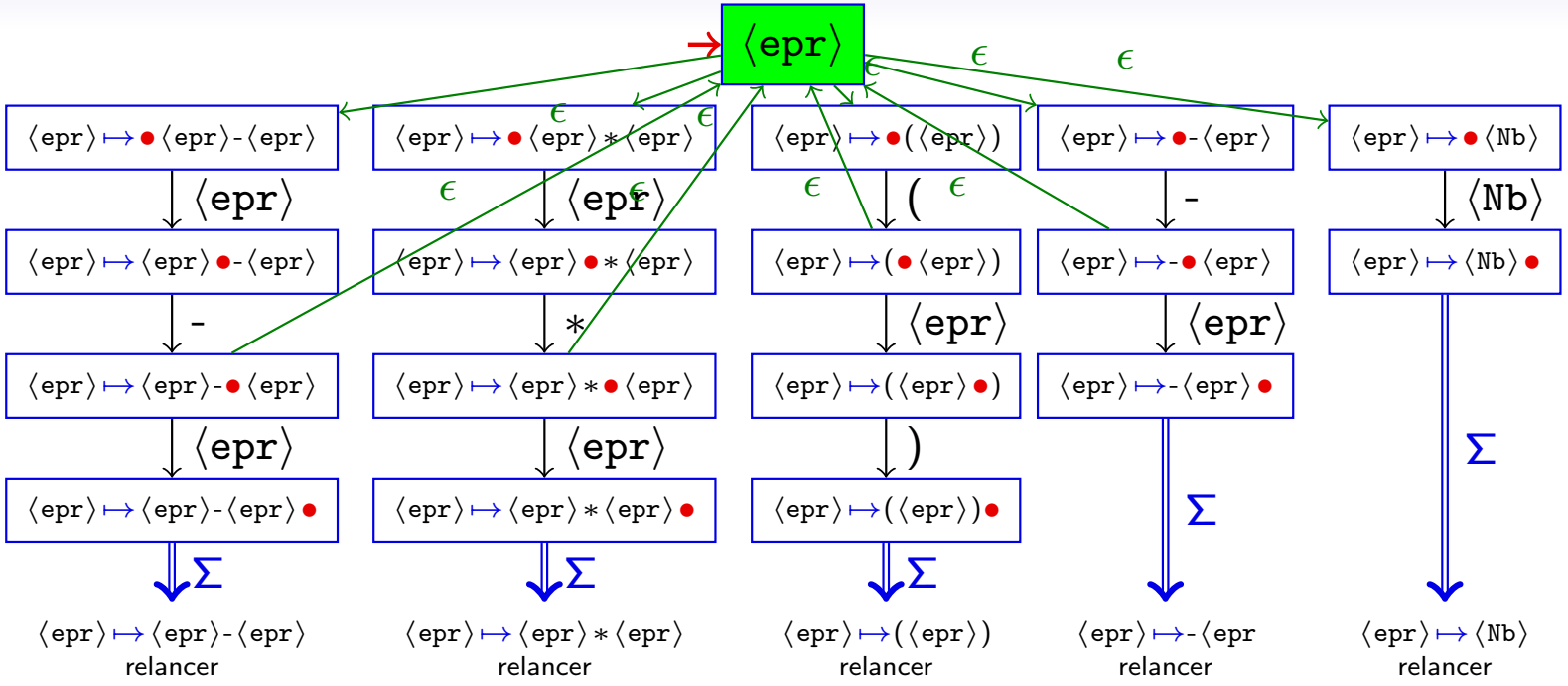


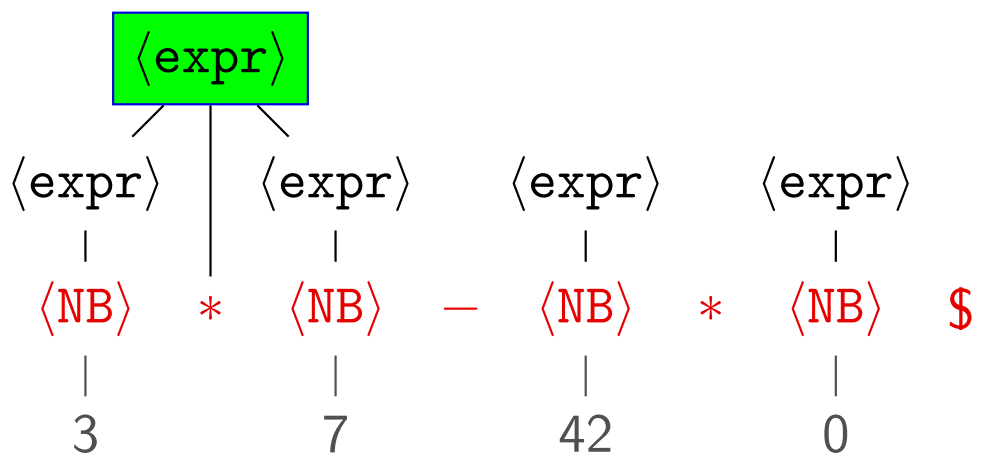
Seconde tentative en action



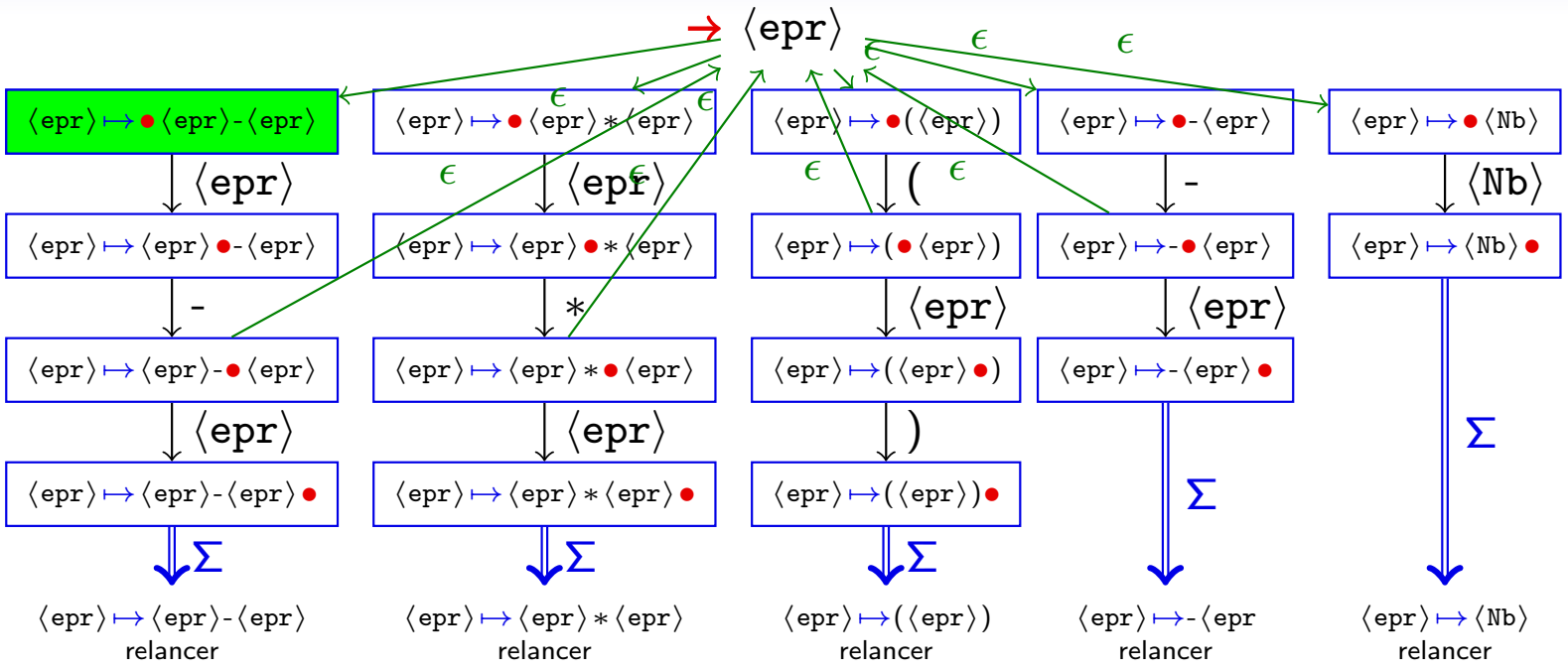


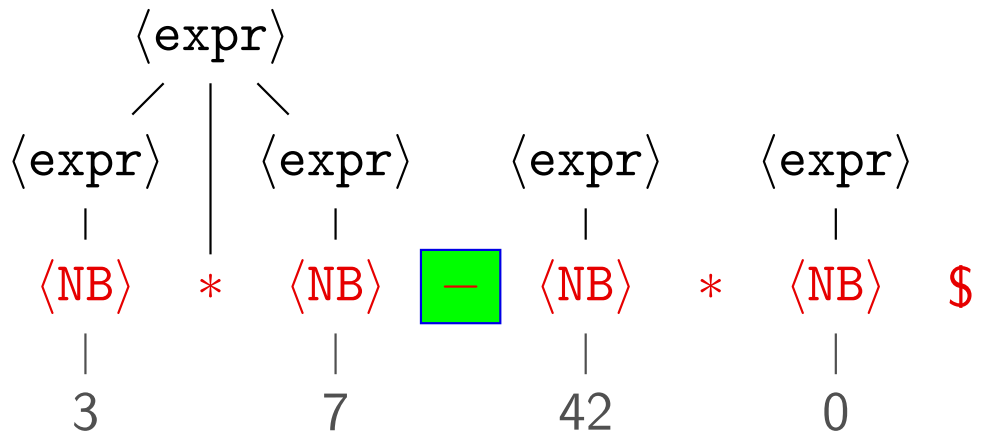
Seconde tentative en action



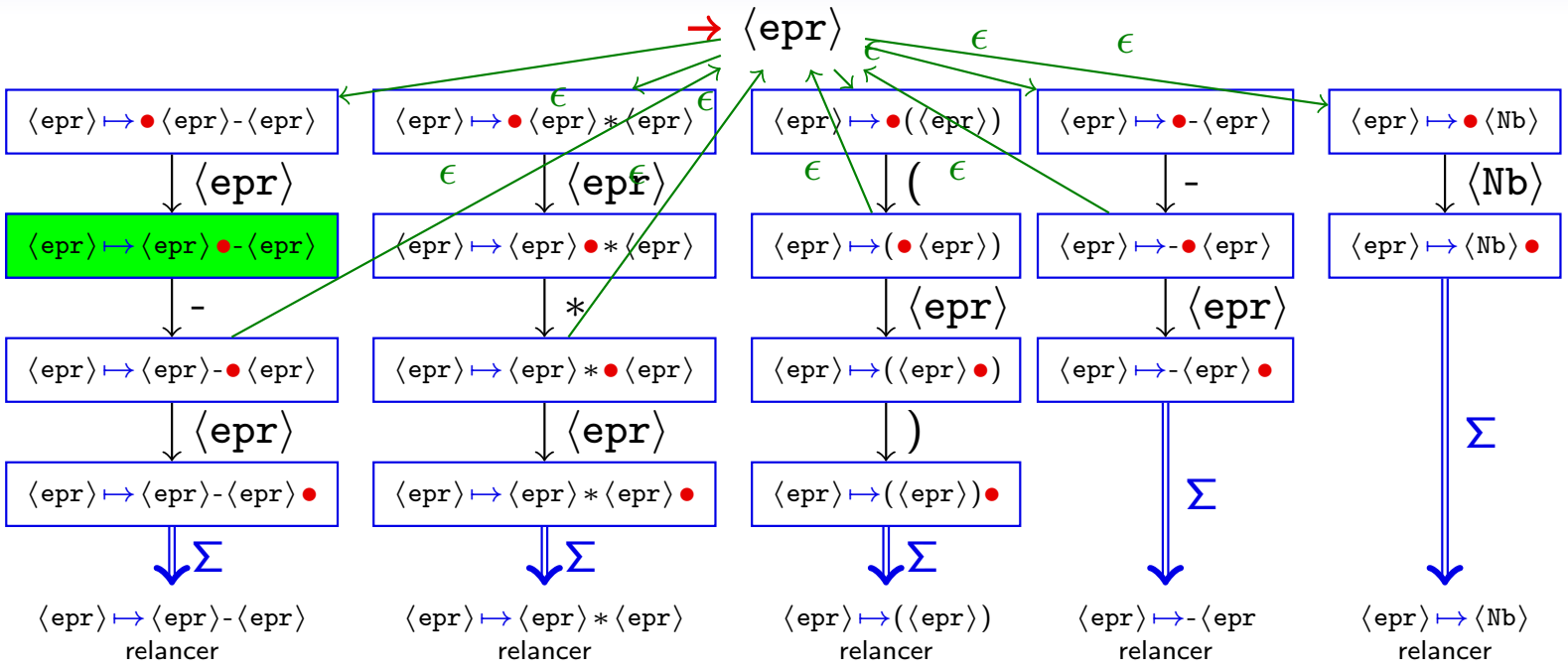


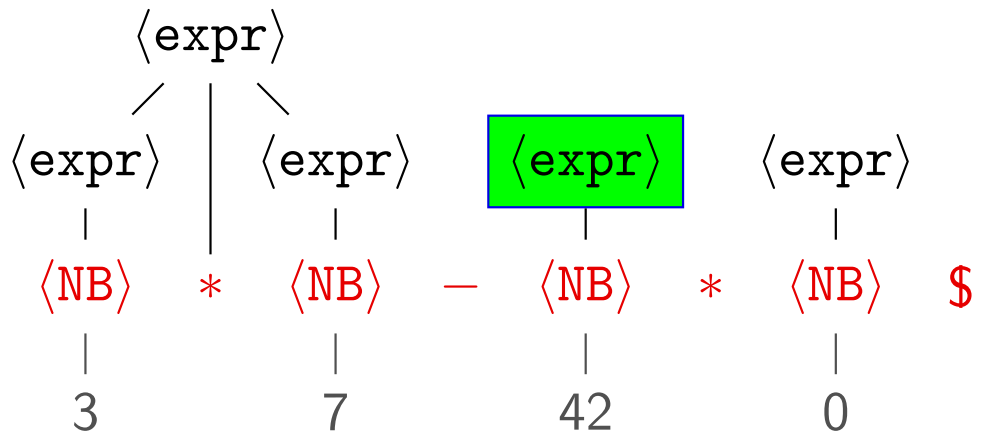
Seconde tentative en action



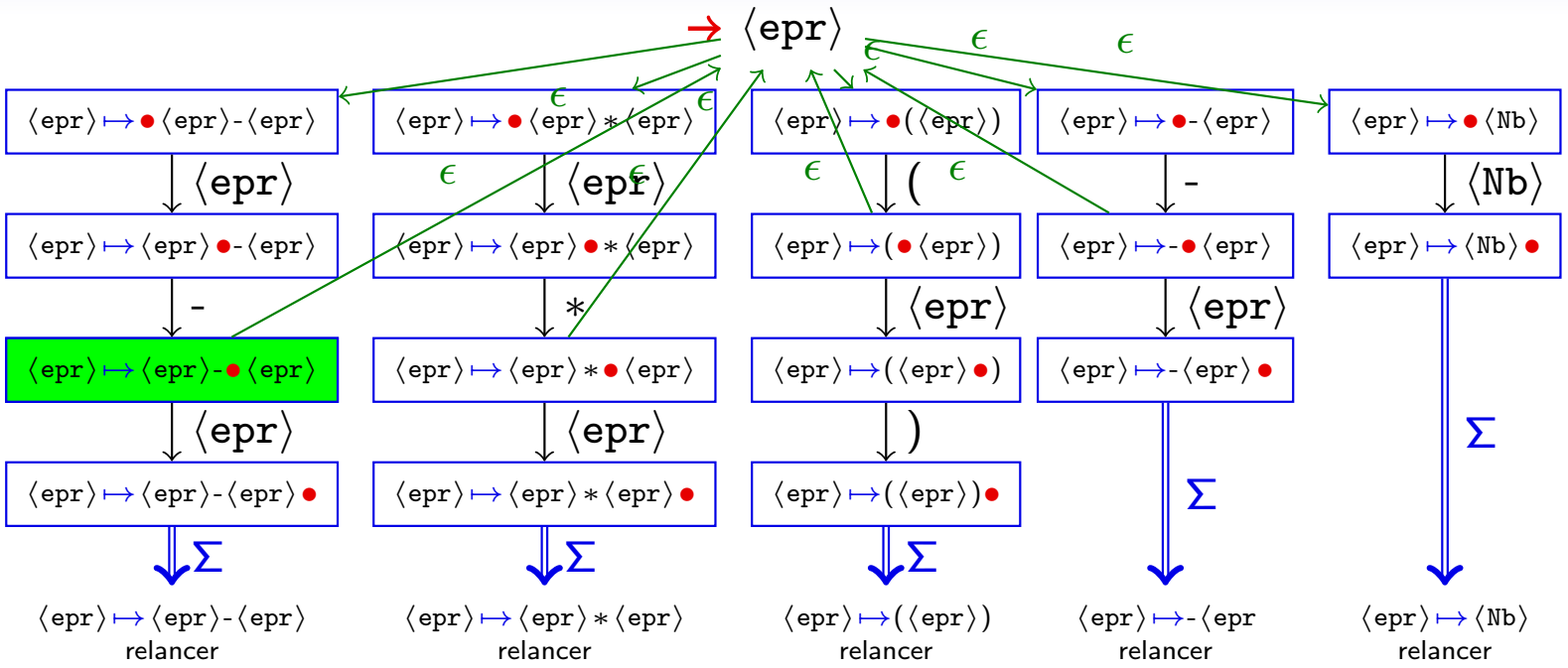


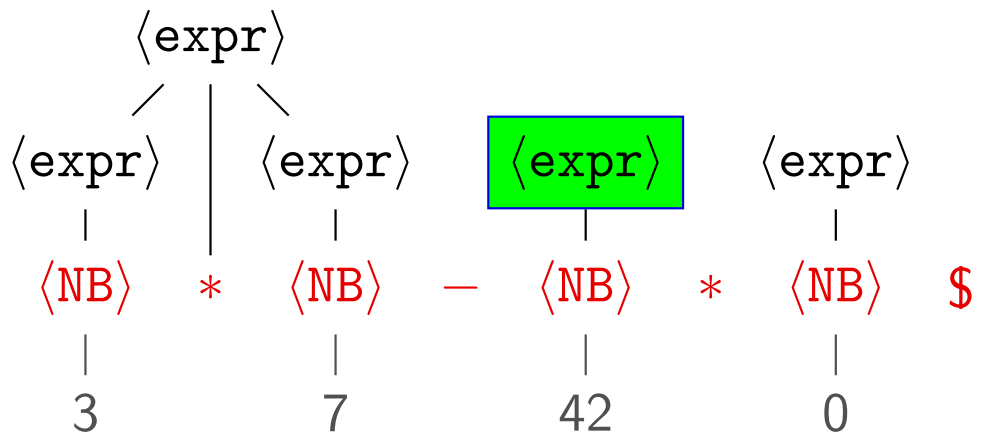
Seconde tentative en action



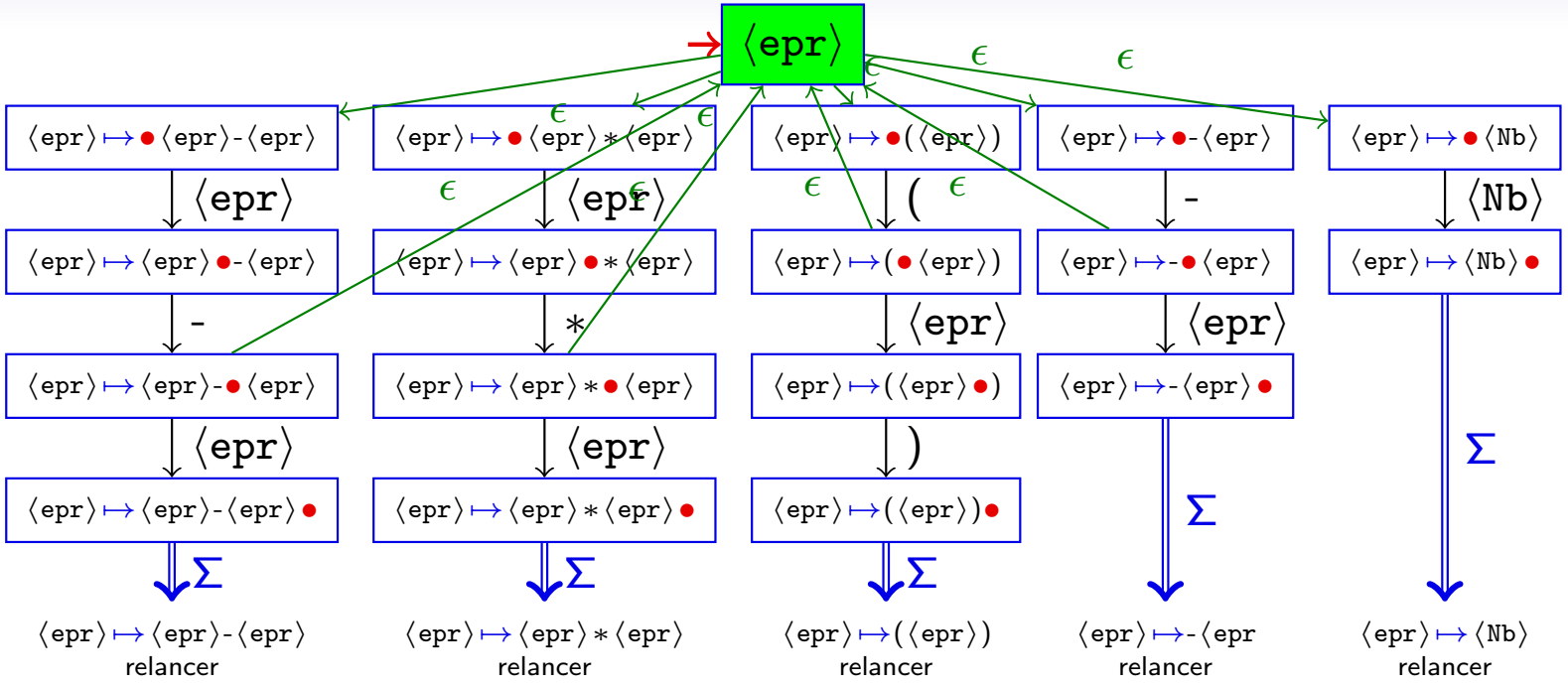


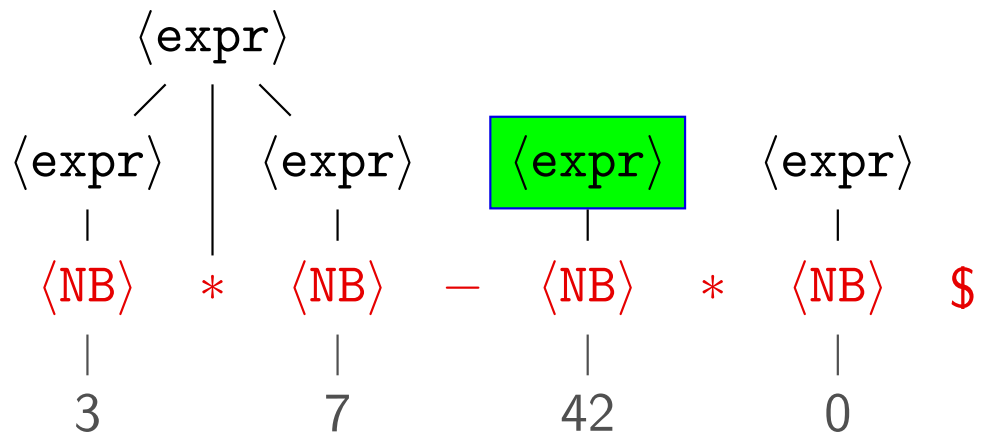
Seconde tentative en action



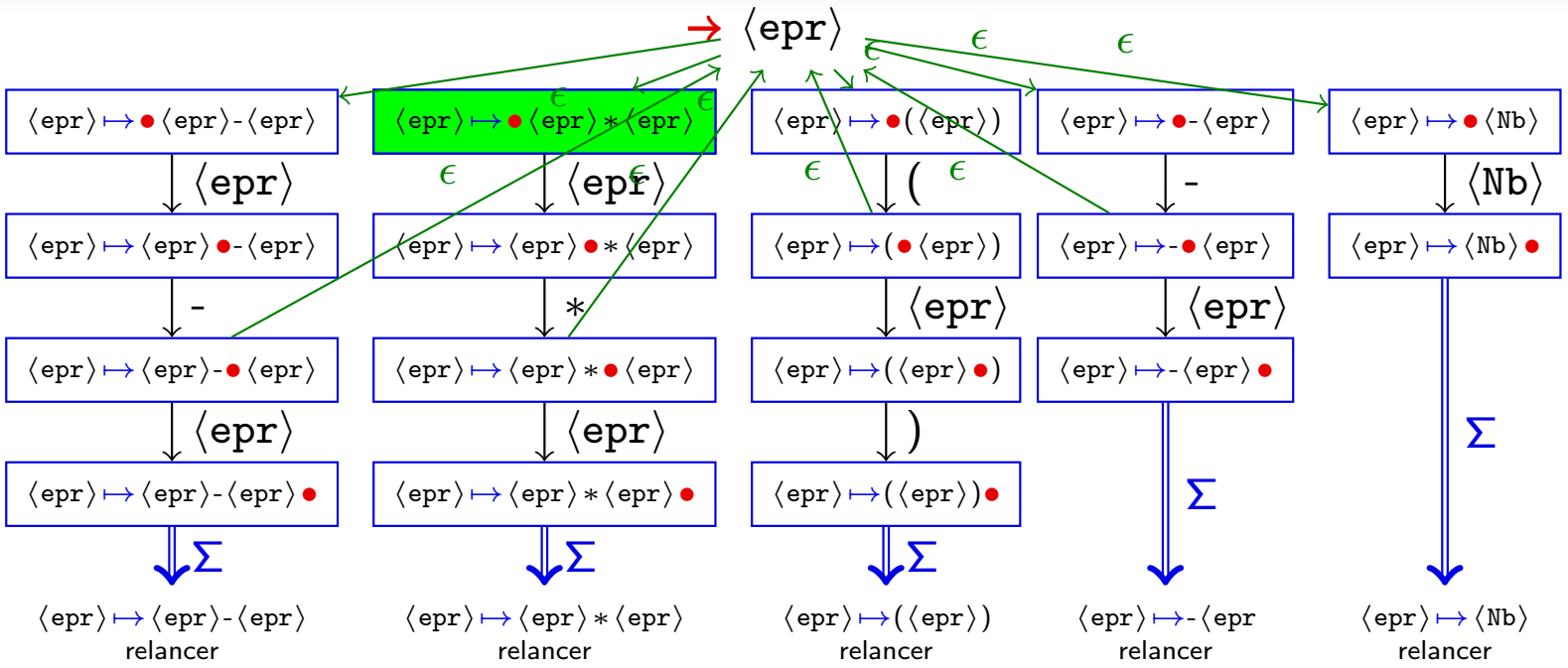


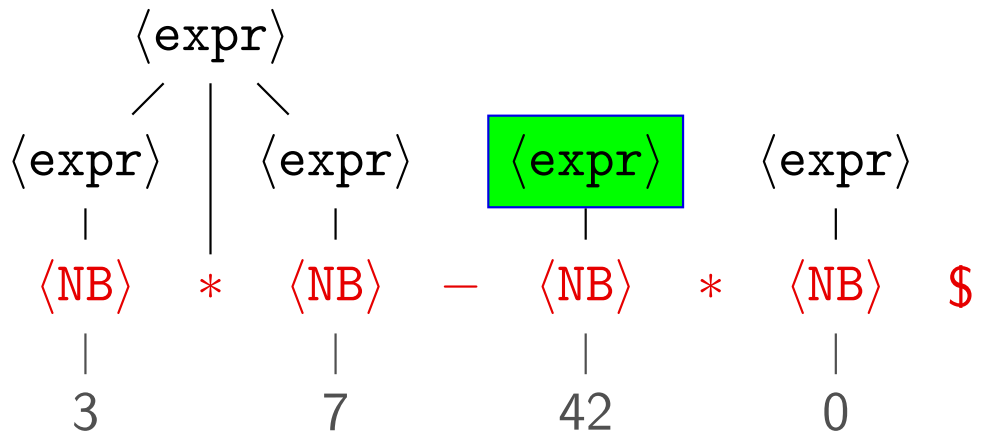
Seconde tentative en action



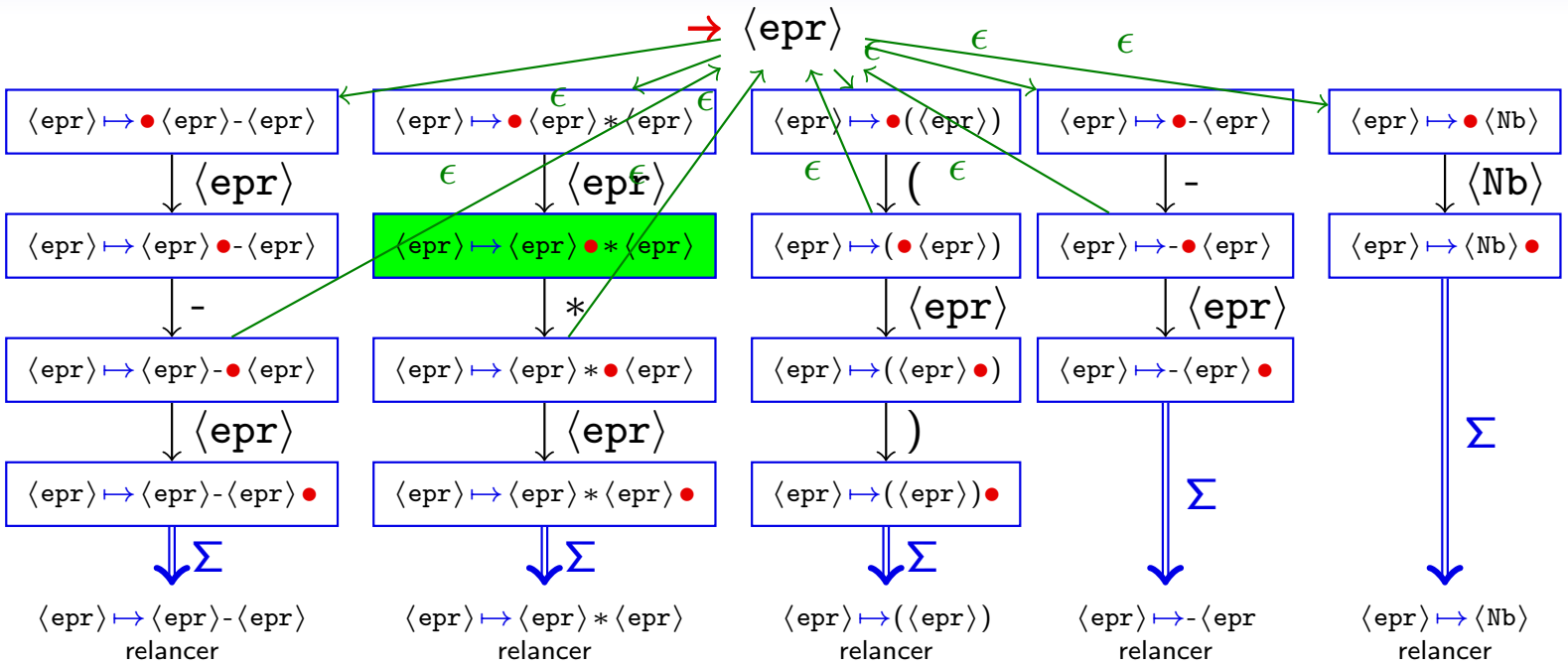


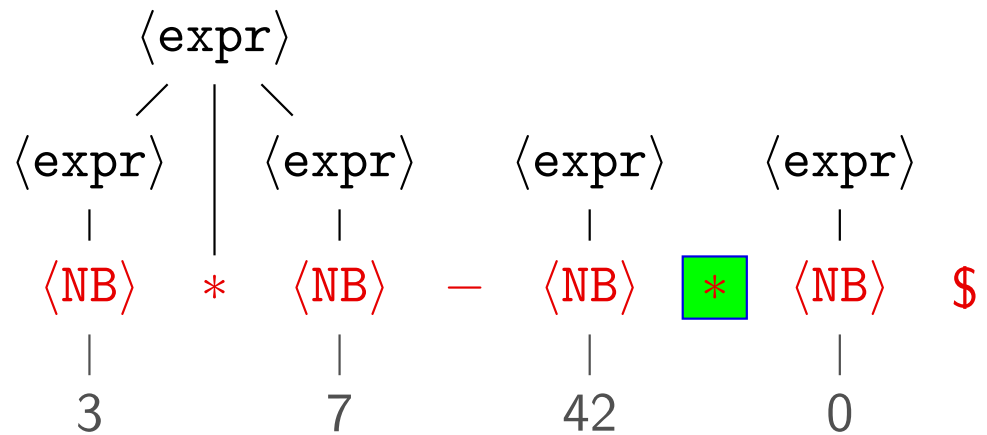
Seconde tentative en action



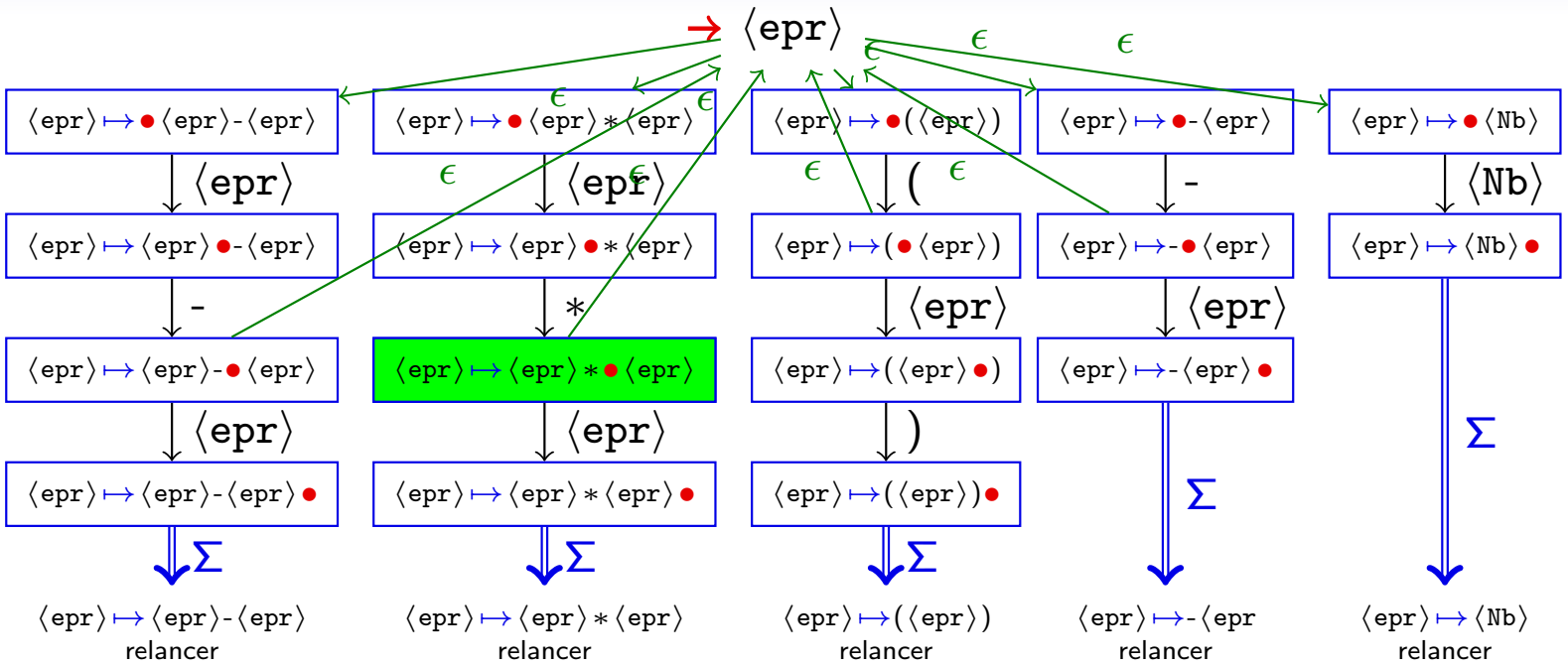


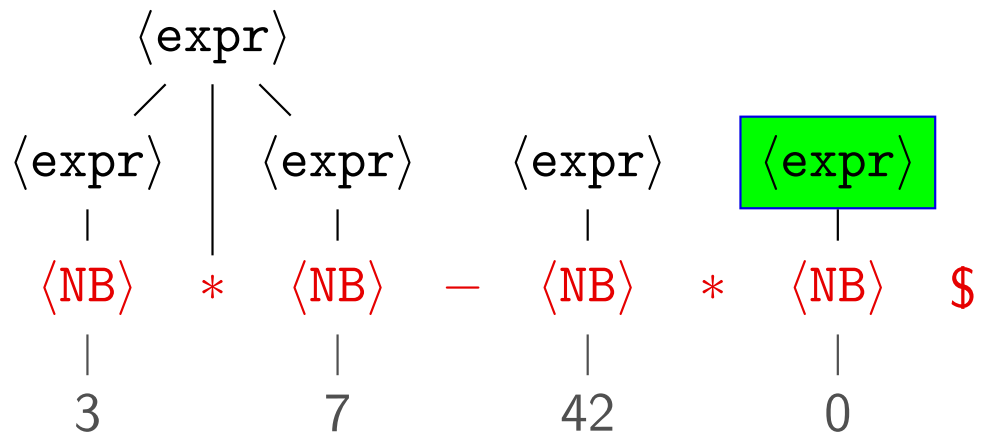
Seconde tentative en action



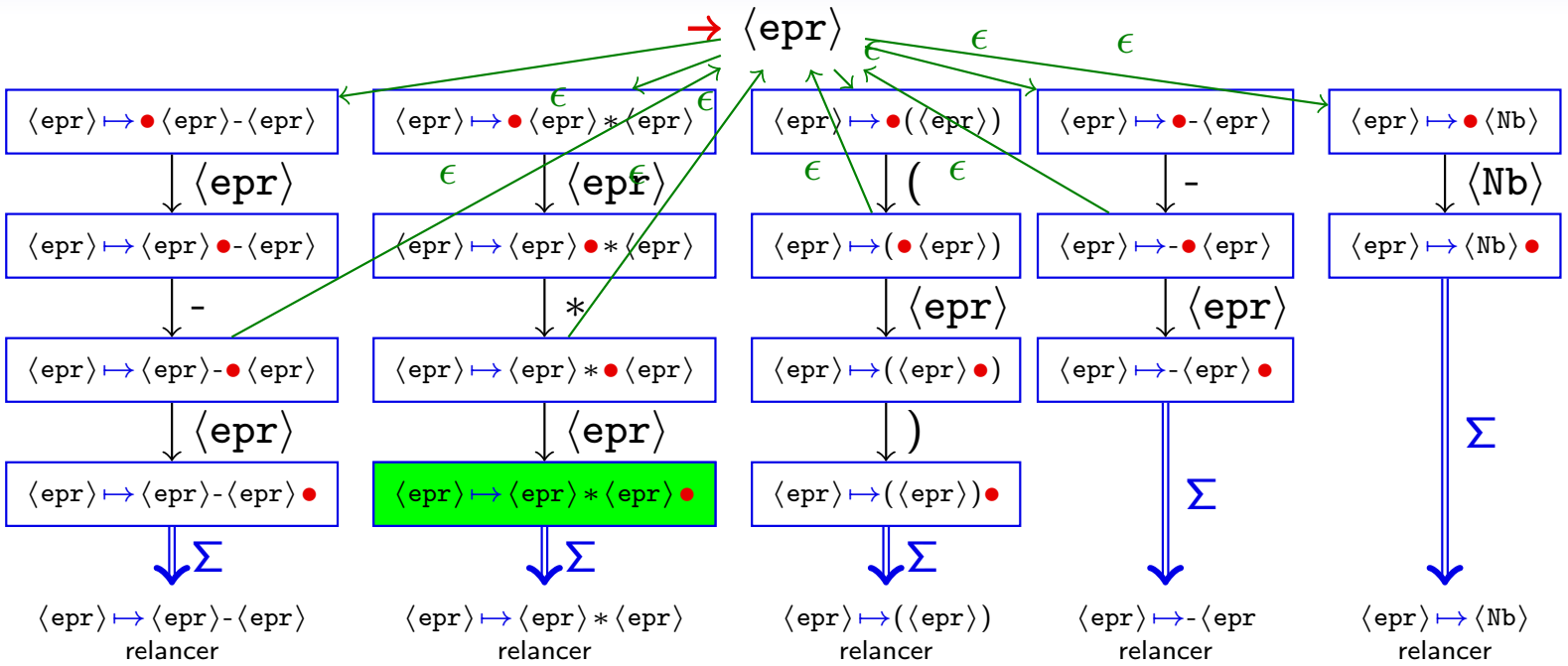


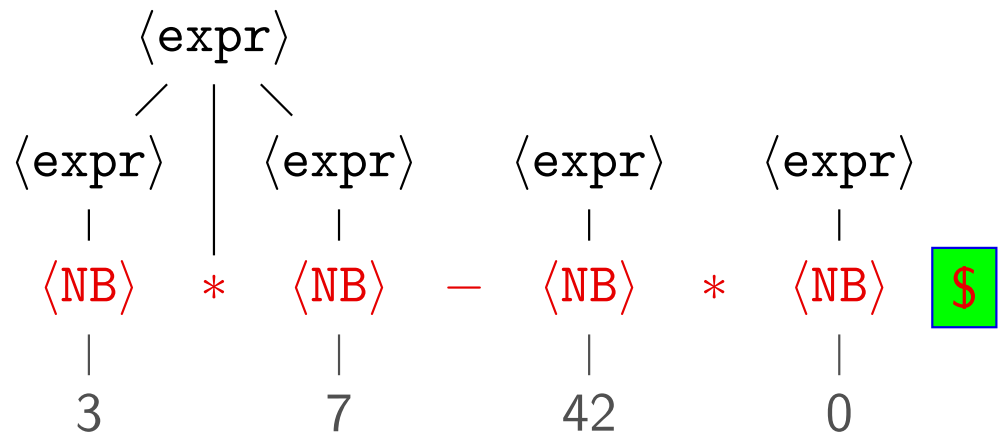
Seconde tentative en action



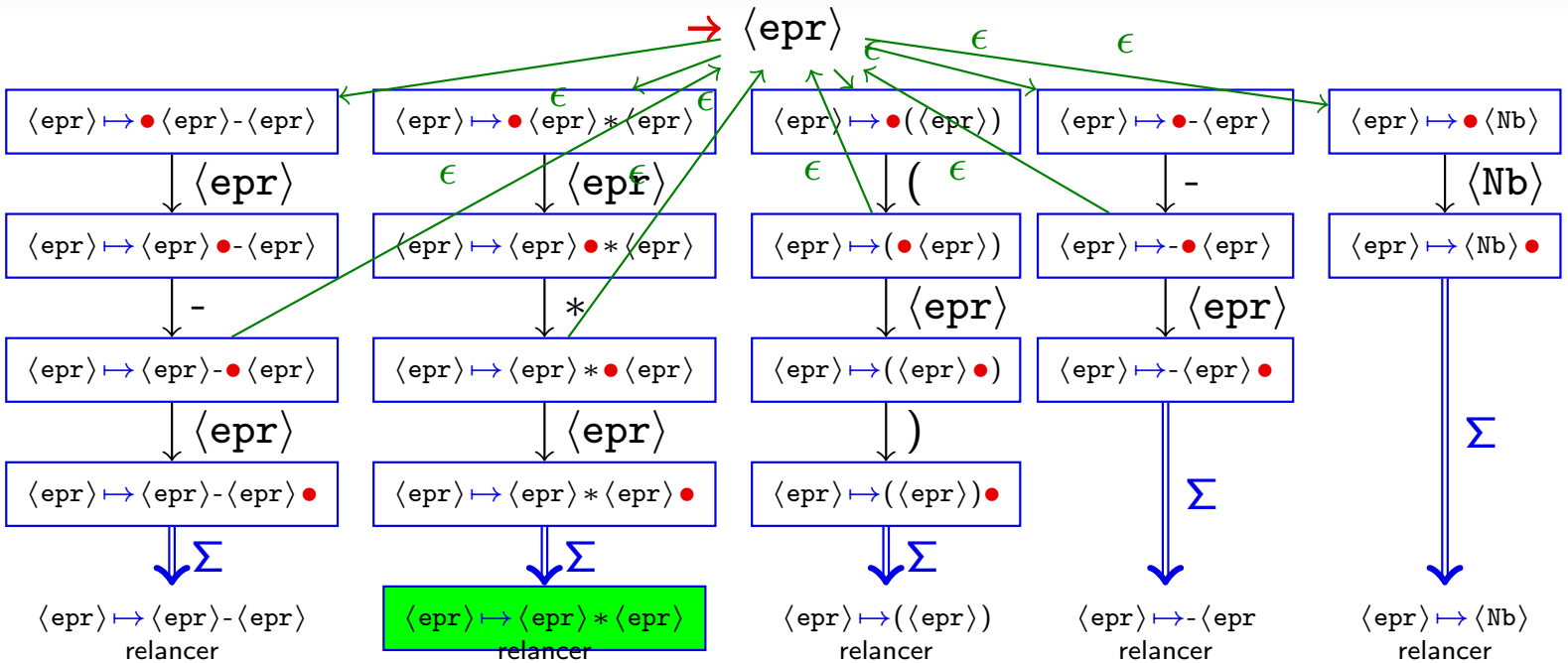


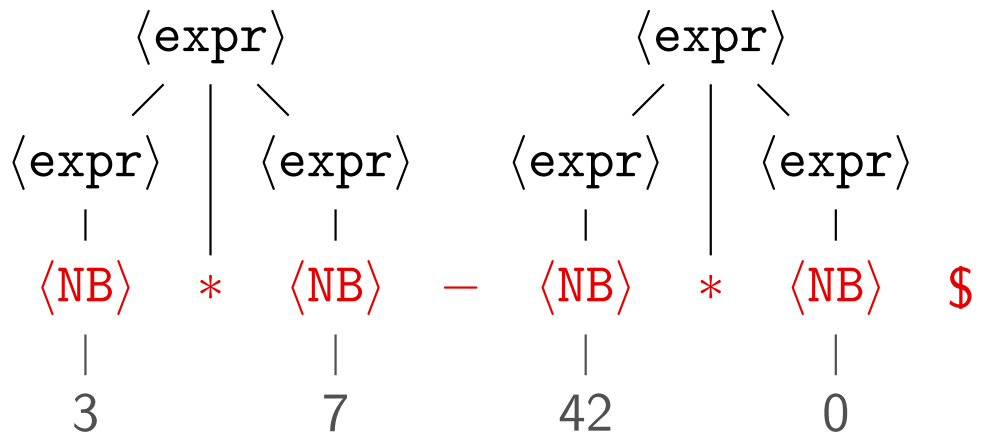
Seconde tentative en action



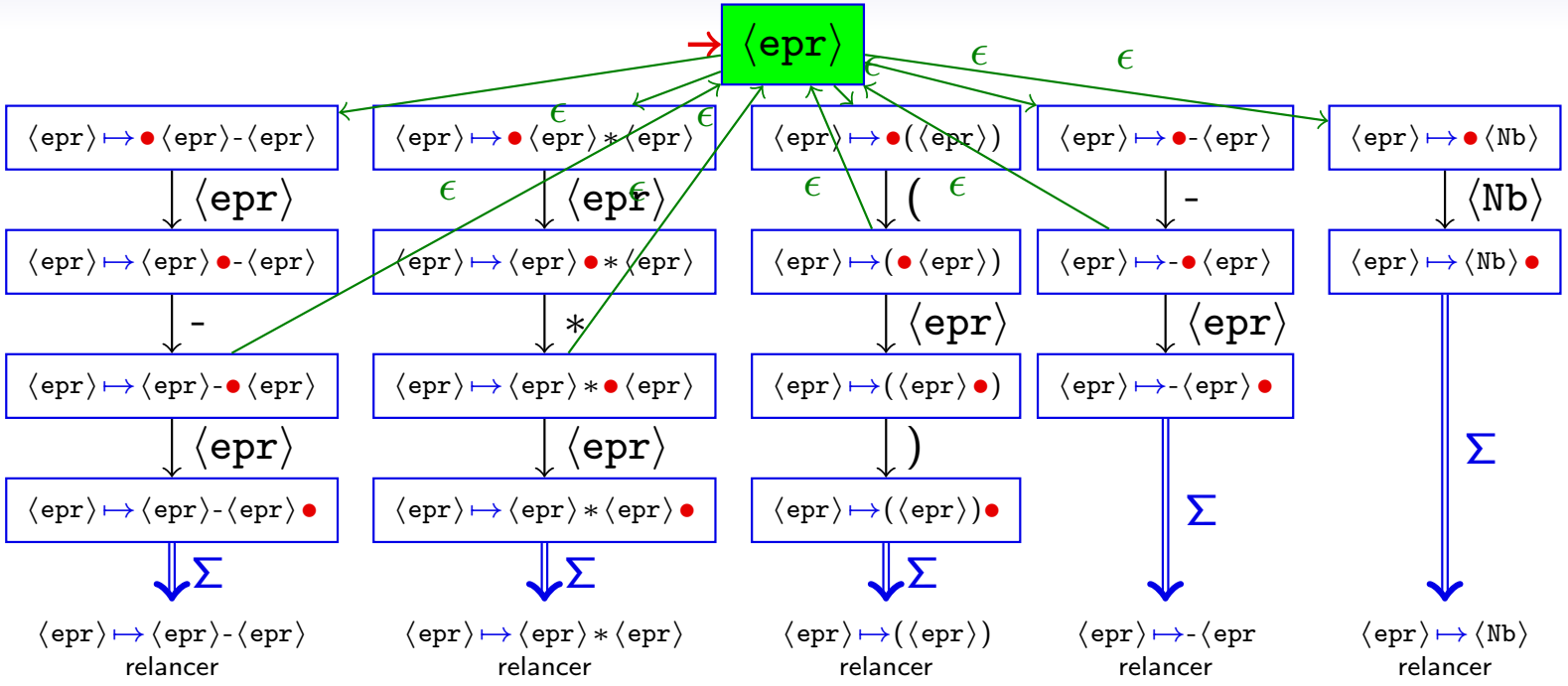


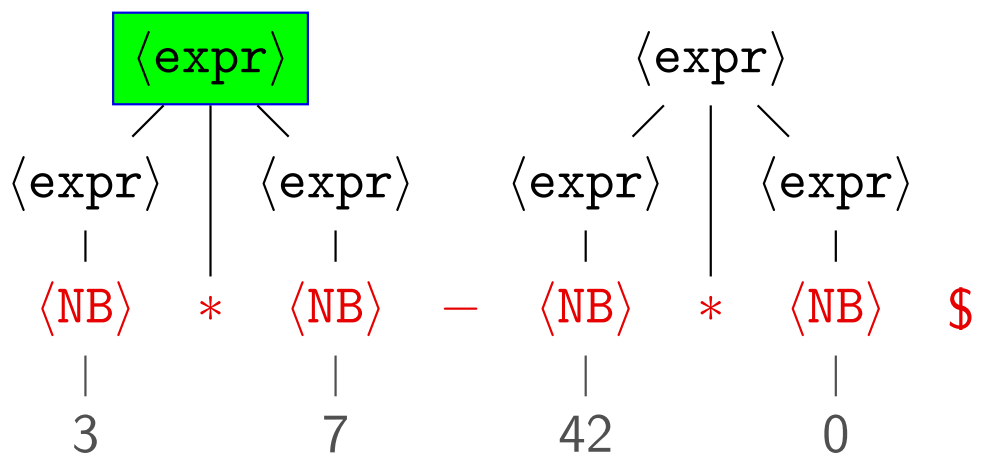
Seconde tentative en action



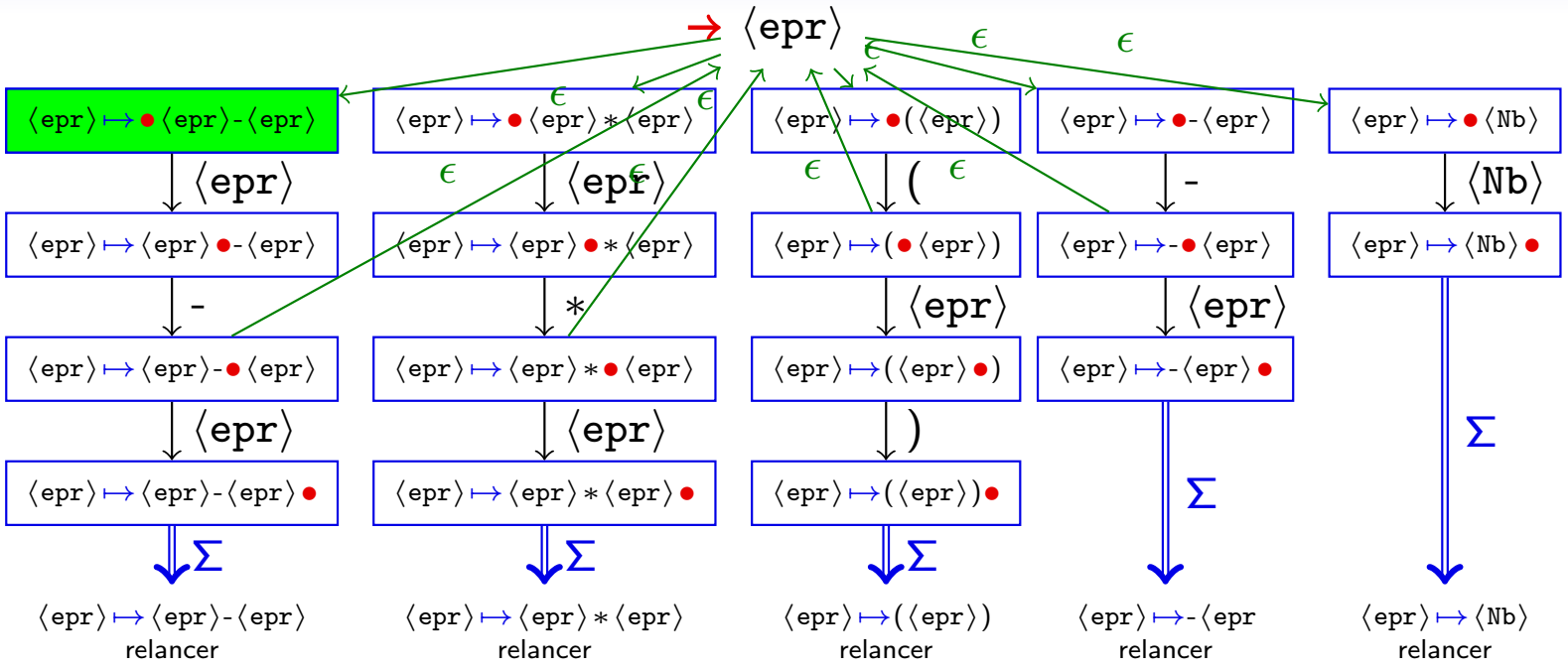


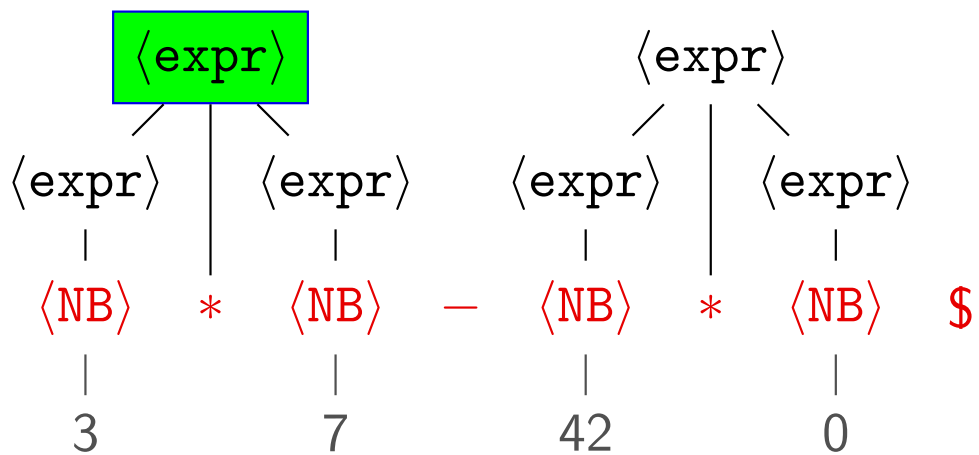
Seconde tentative en action



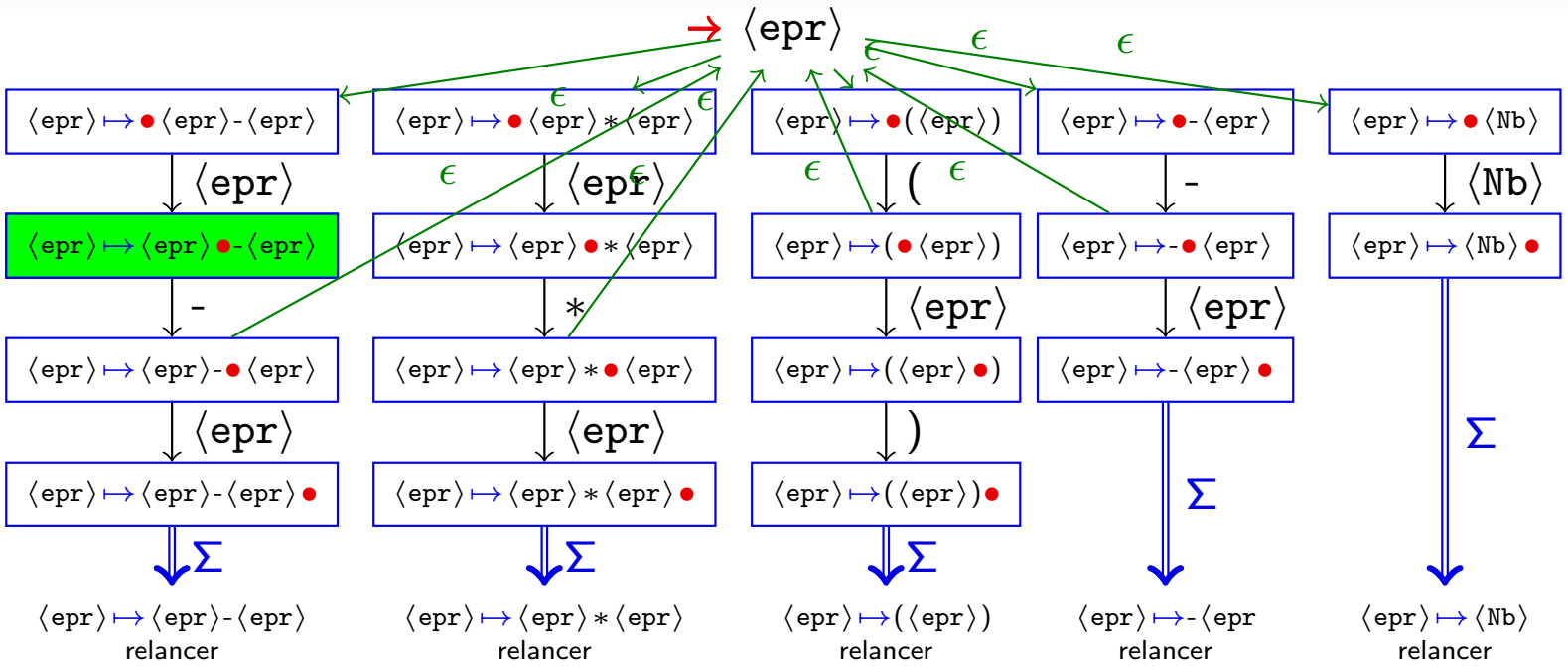


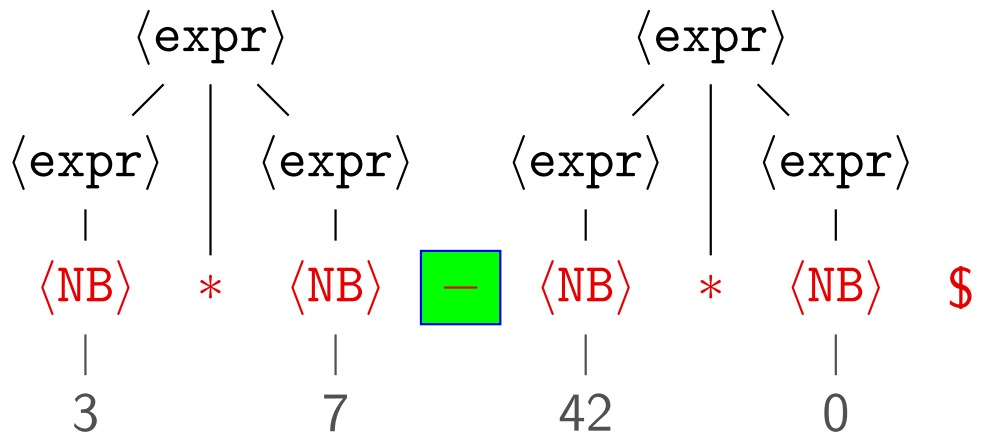
Seconde tentative en action



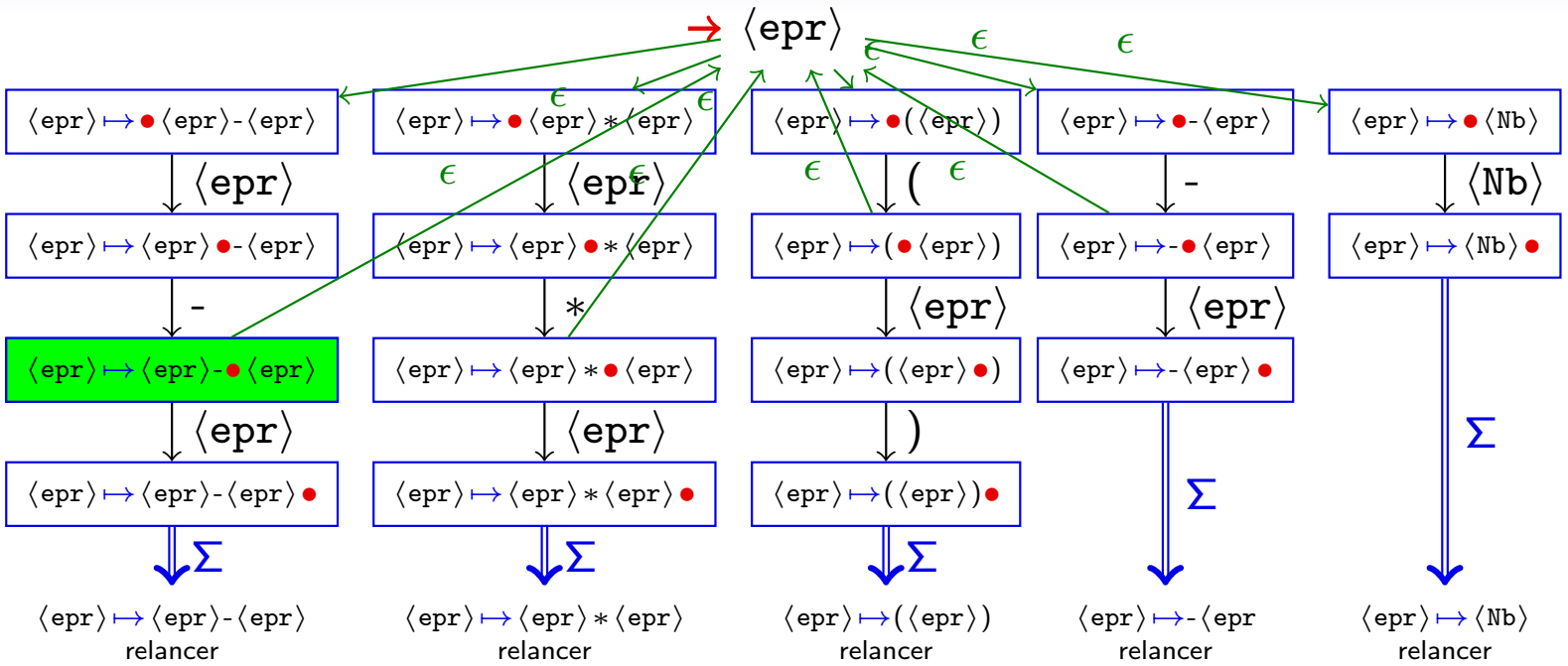


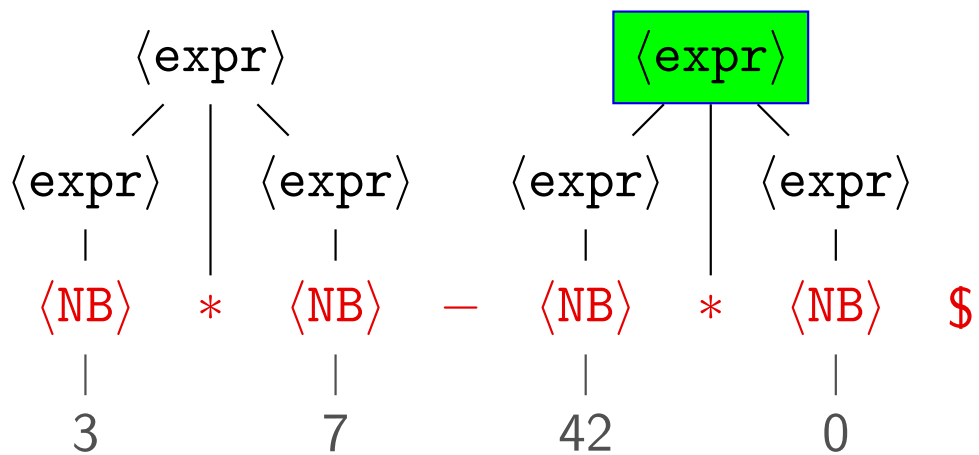
Seconde tentative en action



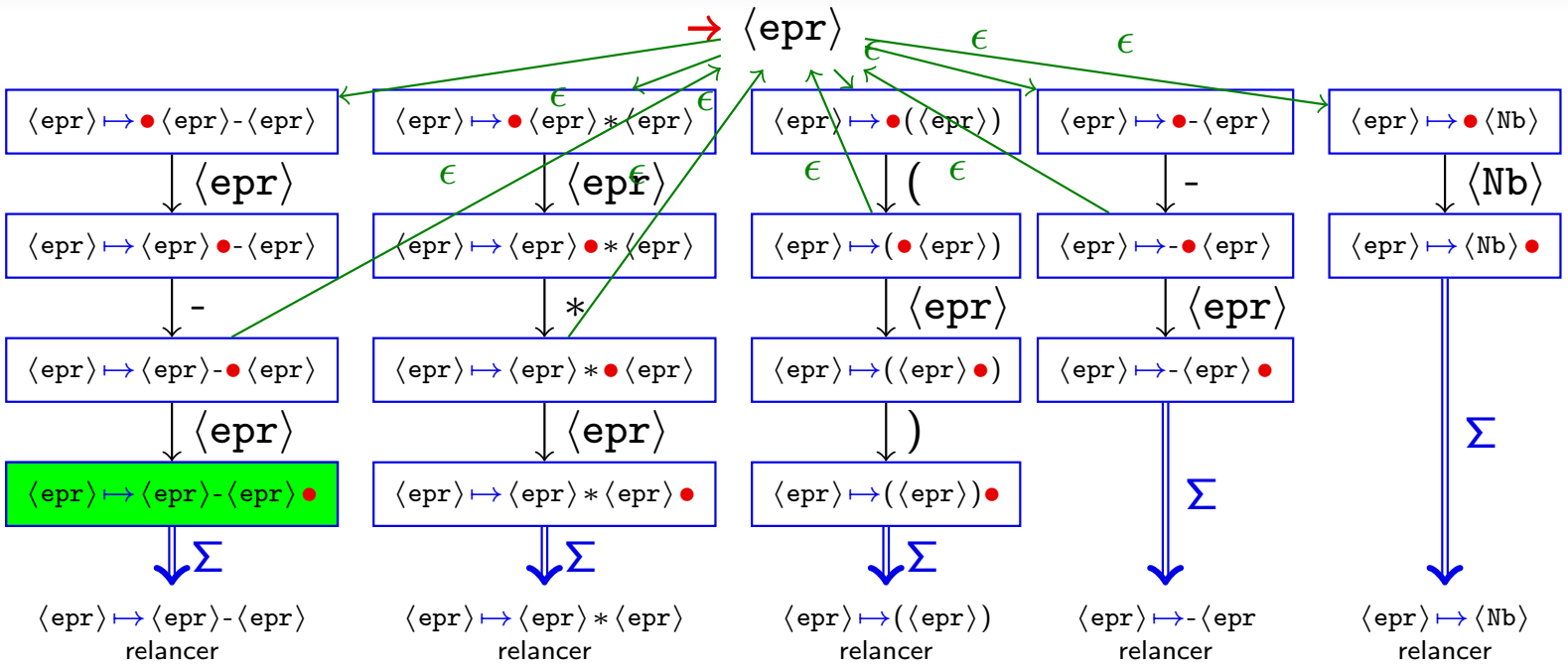


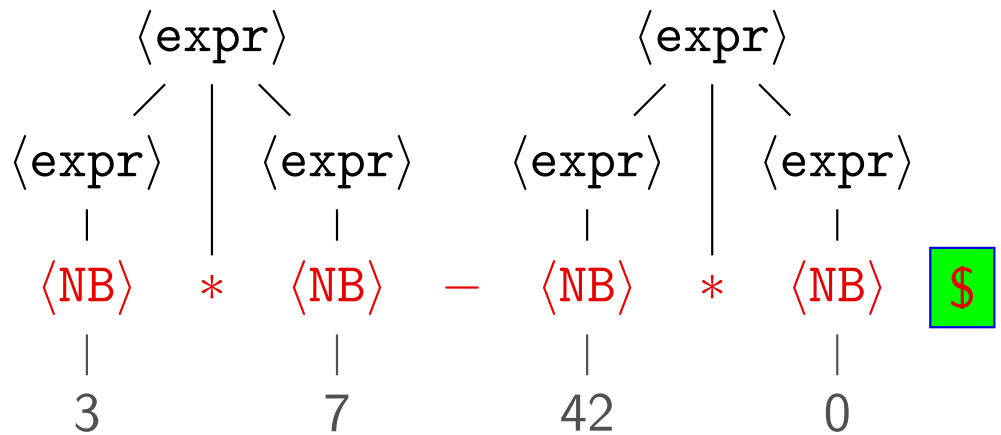
Seconde tentative en action



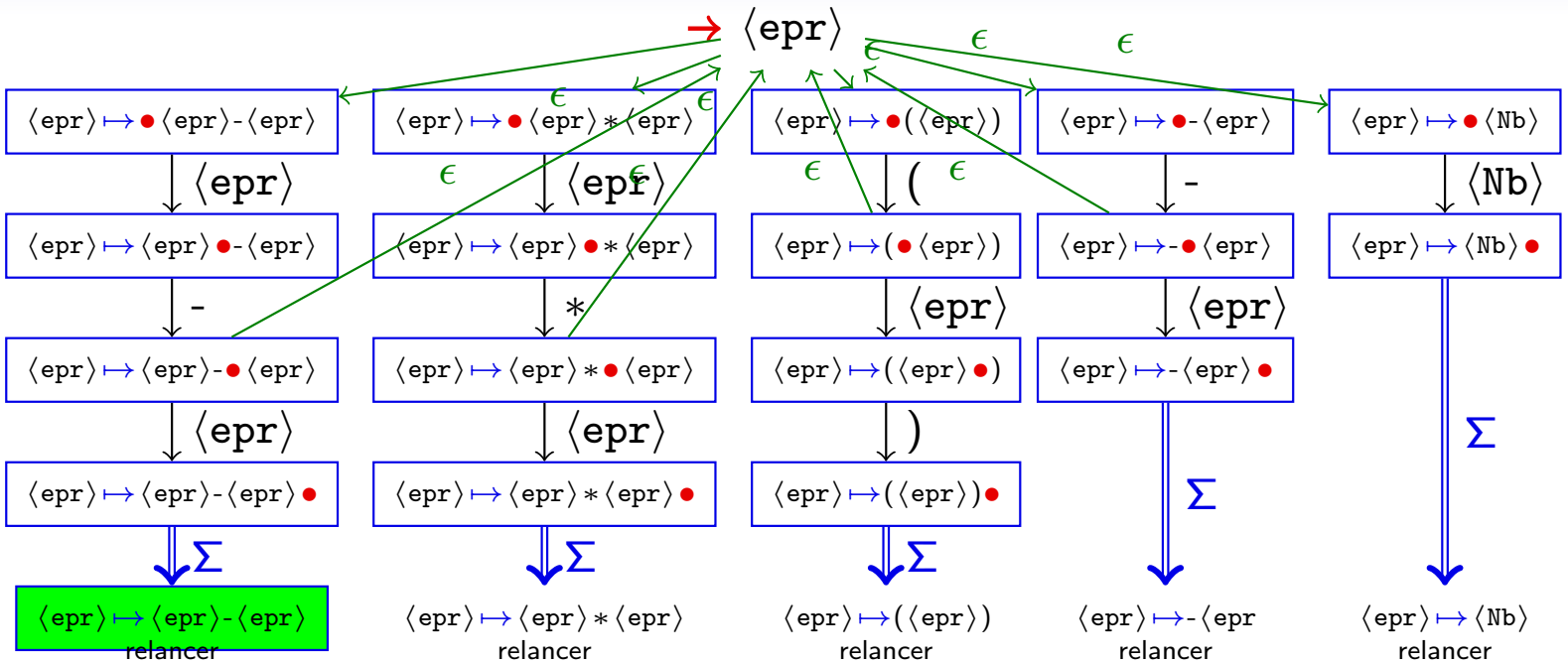


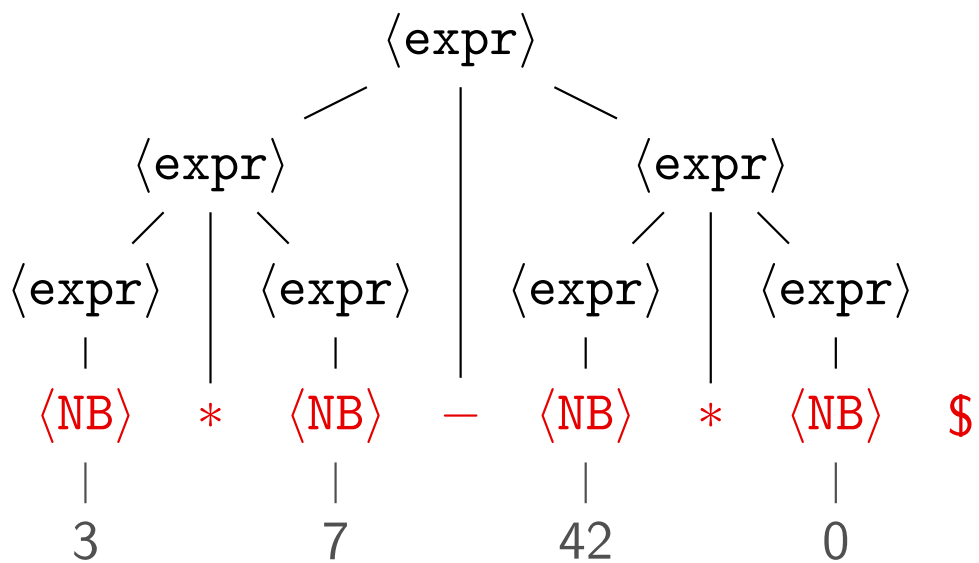
Seconde tentative en action





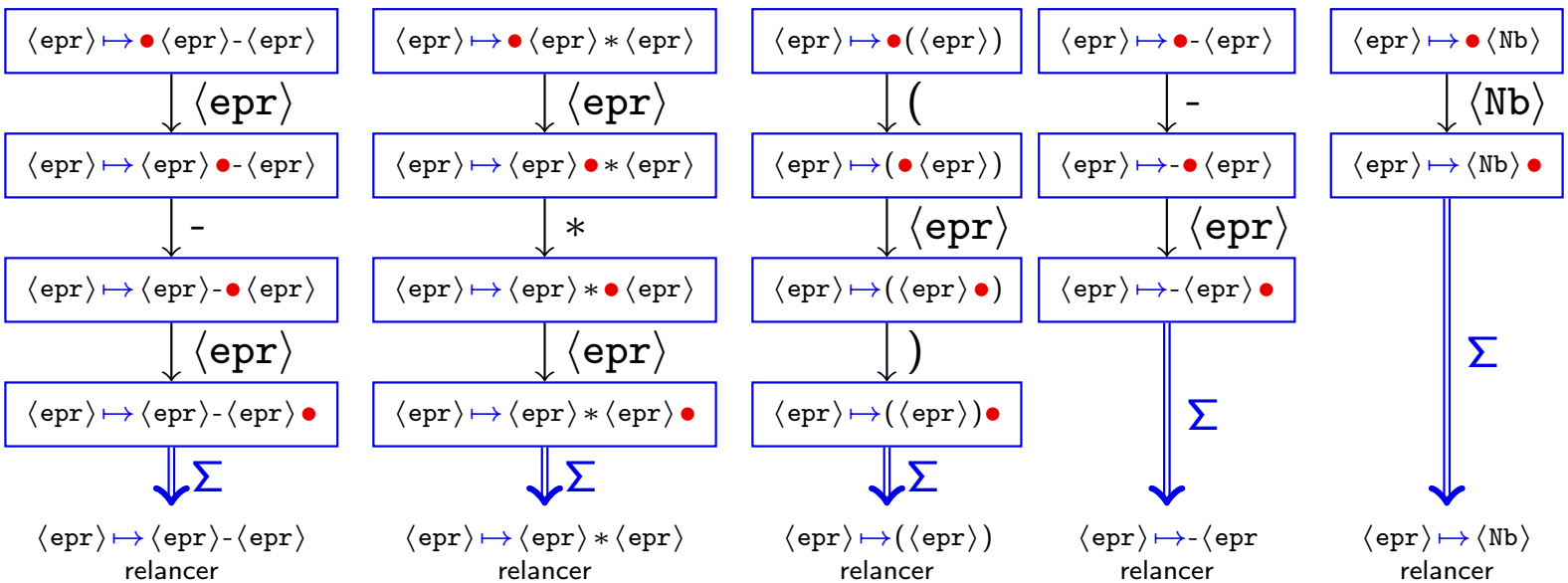
Seconde tentative en action





Quels ϵ-transitions ?

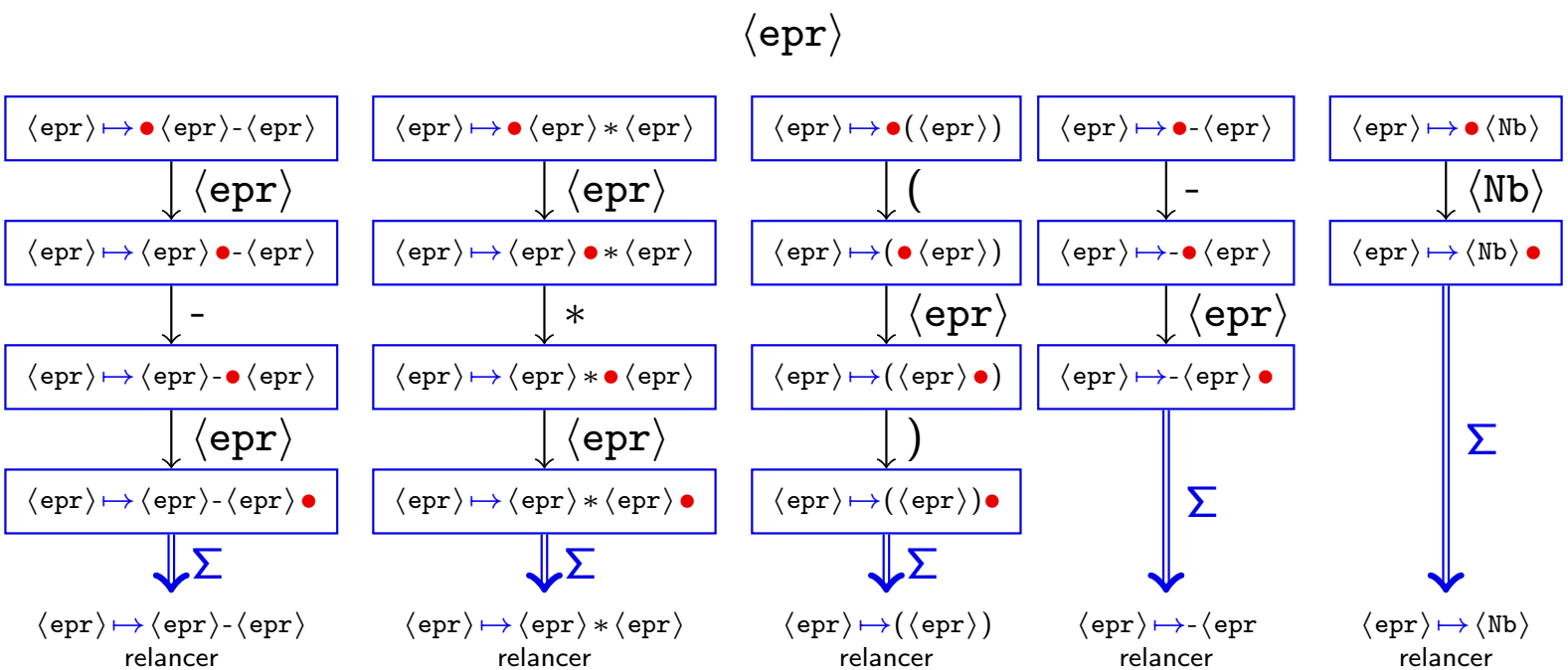
Concretement



Quels ϵ-transitions ?

Concretement

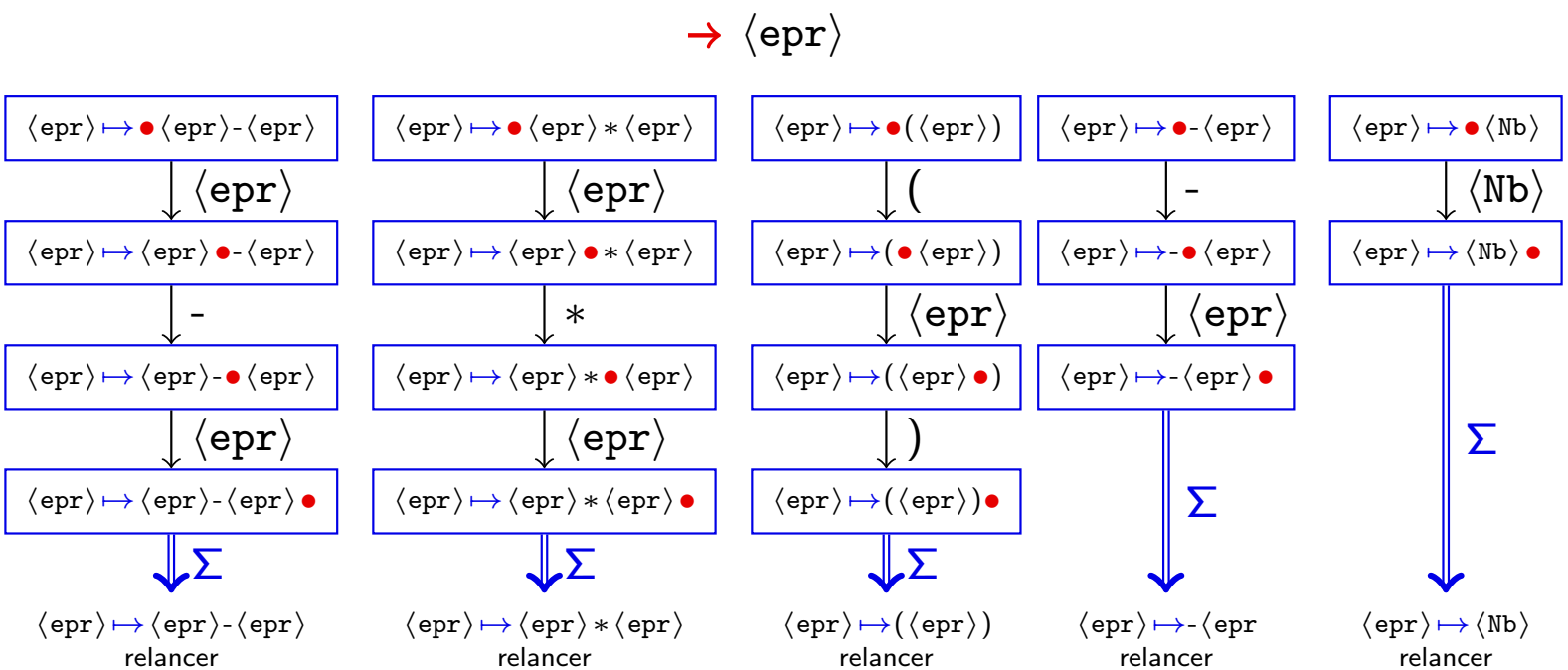
- ajouter un états pour chaque non terminal N,



Quels ϵ-transitions ?

Concretement

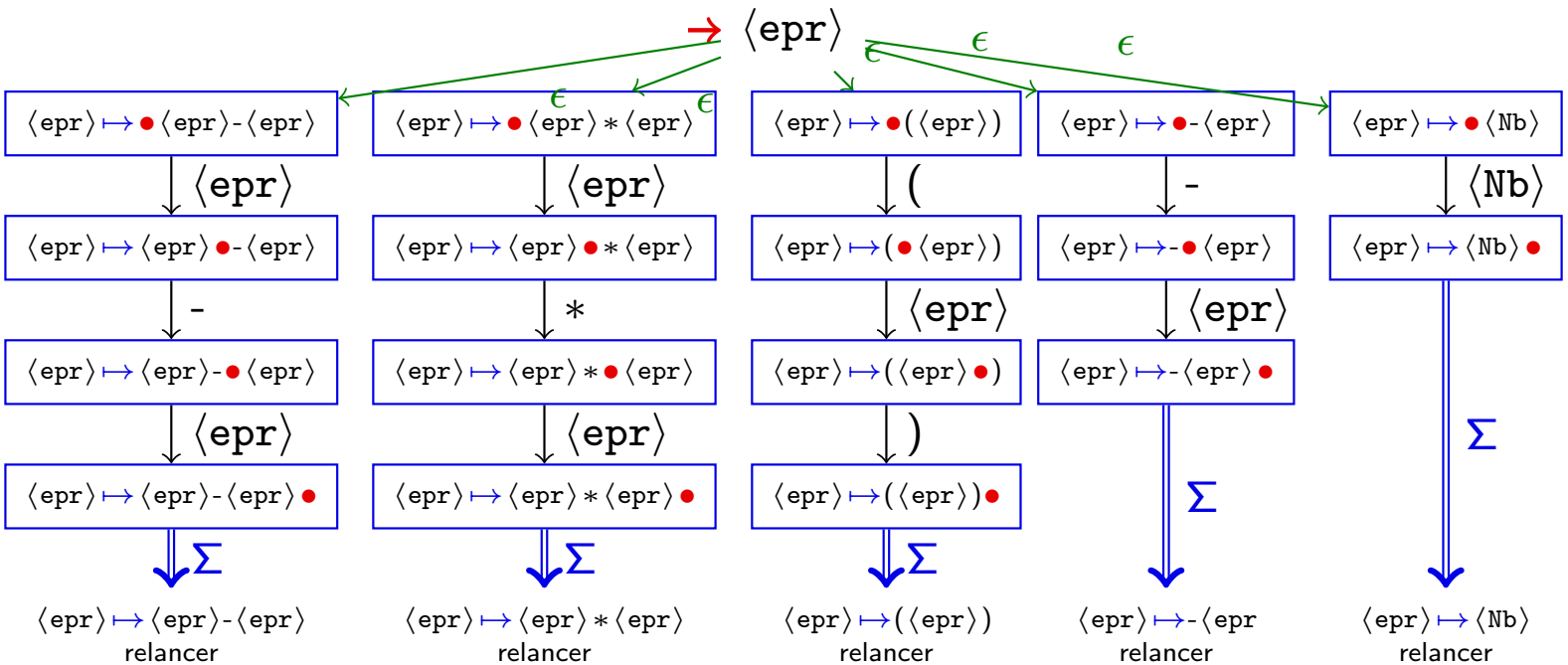
- ajouter un états pour chaque non terminal N,
- seul le principal est un état initial,



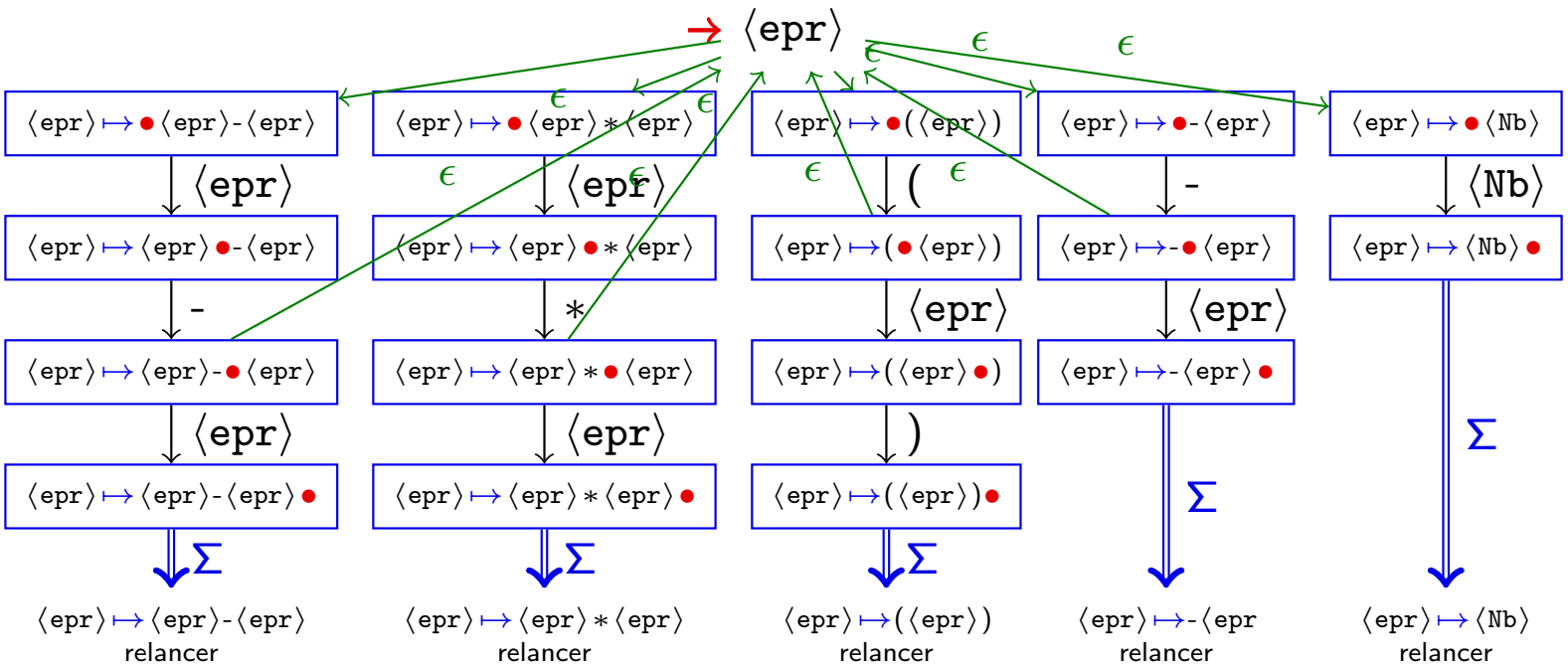
Quels ϵ-transitions ?

Concretement

- ajouter un états pour chaque non terminal N,
- seul le principal est un état initial,
- ajouter des ϵ-transitions vers les rêgles,



Quels ϵ-transitions ?

Concretement

- ajouter un états pour chaque non terminal N,
- seul le principal est un état initial,
- ajouter des ϵ-transitions vers les rêgles,
- pour chaque rêgle de la forme $N \mapsto w_1 \bullet N' w_2$, ajouter une ϵ-transition vers N'