Analyse syntaxique

Gilles Dubochet and Martin Odersky

6 et 13 novembre 2006
version 1.4

Plan du cours

Analyse syntaxique d'une grammaire non-contextuelle

Les grammaires régulières ne peuvent pas exprimer l'imbrication.

Les grammaires non-contextuelles ne peuvent pas être reconnues par des machines à états finis.

Que se passe-t-il si l'on essaie quand même ?

Exemple naÔf : Reconnaître une grammaire non-contextuelle

La grammaire `A = "a" A "c" | "b".` est reconnue par :

```
if (char == "a") {
  next
  if (char == A) next else error;
  if (char == "c") next else error;
} else if (char == "b") next
  else error;
```

Le non-terminal a été traité comme un terminal, ce qui est faux !

Pour dériver un analyseur syntaxique d'une grammaire non-contextuelle écrite dans le style EBNF :

- Introduire une fonction `def A: Unit` pour chaque non-terminal `A`.
  - Celle-ci reconnaît les sous-phrases dérivées de `A`, ou émet une erreur si aucun `A` n'a été trouvé.
- Traduire toutes les expressions régulières des membres droits d'une production comme précédemment, sauf ...
- un non-terminal `B` se traduit maintenant par un appel à la fonction `B`.
  - La récursivité dans la grammaire se traduit naturellement par la récursivité dans l'analyseur syntaxique.

Cette technique pour écrire des analyseurs syntaxiques est appelée analyse syntaxique par descente récursive (en anglais *recursive descent parsing*) ou analyse prédictive.

Exemple : Reconnaître une grammaire non-contextuelle

La grammaire `A = "a" A "c" | "b".` conduit maintenant au ´reconnaisseur[a] suivant :

```
def A: Unit = {
  if (char == "a") {
    next
    A
    if (char == "c") next else error
  } else if (char == "b") next else error
}
```


Analyseurs lexicaux et syntaxiques

La plupart des compilateurs ont à la fois un analyseur lexical pour la syntaxe lexicale et un analyseur syntaxique pour la syntaxe non-contextuelle.

Composant	Entrée	Sortie
Analyseur lexical	Caractères	Lexèmes
Analyseur syntaxique	Lexèmes	Arbre syntaxique

Avantages : séparation des objectifs, meilleure modularité.

Exemple : Analyseur syntaxique EBNF

Nous écrivons l'analyseur syntaxique comme une extension de l'analyseur lexical :

Exemple : Analyseur syntaxique EBNF

```
class EBNFParser (in: InputStream) extends EBNFScanner(in) {
  nextToken;
```

Chaque production est ensuite traduite suivant le schéma donné. La production pour analyser toute une grammaire s'écrit comme ceci :

```
  /** syntax = {production} <eof>. */
  def syntax: Unit =
    while (token != EOF) production;
```

`EBNFParser` continue sur la page suivante ...


```
  /** production = identifier "=" expression ".". */
  def production: Unit = {
    if (token == IDENT) nextToken else error("illegal␣start␣of␣production")
    if (token == EQL) nextToken else error("'='␣expected")
    expression
    if (token == PERIOD) nextToken else error("'.'␣expected")
    }
  /** expression = term {"|" term}. */
  def expression: Unit = { term; while (token == BAR) { nextToken; term } }
  /** term = {factor}. */
  val firstFactor = List(IDENT,LITERAL,LPAREN,LBRACK,LBRACE)
  def term: Unit = while (firstFactor.contains(token)) factor
  /** factor = identifier | string | "(" expression ")"
             | "[" expression "]" | "{" expression "}". */
  def factor: Unit = token match {
    case IDENT => nextToken
    case LITERAL => nextToken
    case LPAREN =>
      nextToken; expression
      if (token == RPAREN) nextToken else error("')'␣expected")
    case LBRACK =>
      nextToken; expression
      if (token == RBRACK) nextToken else error("']'␣expected")
    case LBRACE =>
      nextToken; expression
      if (token == RBRACE) nextToken else error("'}'␣expected")
    case _ => error("illegal␣start␣of␣factor")
}}
```

Exemple en Java : Analyseur syntaxique EBNF

```
class EBNFParser extends EBNFScanner {
  public Parser (InputStream in) {
    super(in);
    nextToken();
  }
  /* syntax = {production} <eof> */
  public void syntax () {
    while (sym != EOF) {
      production();
  }}
  /* production = identifier "=" expression "." */
  void production () {
    if (sym == IDENT) nextToken()
    else error("illegal␣startof␣production");
    if (sym == EQL) nextToken()
    else error("'='␣expected");
    expression();
    if (sym == PERIOD) nextToken()
    else error("'.'␣expected");
  }
  /* expression = term {"|" term} */
  void expression () {
    term();
    while (sym == BAR) {
      nextToken();
      term();
  }}
```



```
  /* term = {factor} */
  void term () {
    while (sym == IDENT || sym == LITERAL || sym == LPAREN || sym == LBRACK || sym == LBRACE)
      factor();
  }
  /* factor = identifier | string | "(" expression ")" |
              "[" expression "]" | "{" expression "}" */
  void factor () {
    switch(sym) {
      case IDENT: nextToken(); break;
      case LITERAL: nextToken(); break;
      case LPAREN:
        nextToken()
        expression();
        if (sym == RPAREN) nextToken()
        else error("')'␣expected");
        break;
      case LBRACK:
        nextToken()
        expression();
        if (sym == RBRACK) nextToken()
        else error("']'␣expected");
        break;
      case LBRACE:
        nextToken()
        expression();
        if (sym == RBRACE) nextToken()
        else error("'}'␣expected");
        break;
      default: error("illegal␣start␣of␣factor");
}}}
```


Grammaires analysables par la gauche

Comme auparavant, la grammaire doit être analysable par la gauche pour que ce schéma fonctionne.

Une grammaire est analysable par la gauche aux conditions suivantes :

1. Dans une alternative T_1 | ... | T_n, les termes ne doivent pas avoir de symbole de départ commun.
2. Si une partie exp d'une expression régulière contient la chaîne vide alors exp ne peut être suivi par un symbole qui est aussi un symbole de départ de exp.

Nous formalisons maintenant ce que cela signifie en définissant pour chaque symbole X les ensembles first(X) et follow(X) et l'ensemble nullable.

first, follow et nullable

Étant donné un langage non contextuel analysable par la gauche, on définit :

- `nullable` l'ensemble des non-terminaux qui peuvent dériver la chaîne vide.
- `first(X)` l'ensemble des symboles terminaux qui peuvent commencer les chaînes dérivées de X.
- `follow(X)` l'ensemble des symboles terminaux qui peuvent suivre immédiatement X. C'est-à-dire t $\in$ follow(X) s'il y a une dérivation qui contient X t.

Ou plus formellement :

Définition : `first(X)`, `follow(X)` et `nullable`

Soit $X = Y_1 \ \ldots \ Y_k$, $1 \leq i < j \leq k$, `first(X)`, `follow(X)` et `nullable` sont les plus petits ensembles qui vérifient :

$$\begin{aligned}
\{Y_1,\ldots,Y_k\} \subseteq \texttt{nullable} &\Rightarrow X \in \texttt{nullable} \\
\{Y_1,\ldots,Y_{i-1}\} \subseteq \texttt{nullable} &\Rightarrow \texttt{first}(Y_i) \subseteq \texttt{first}(X) \\
\{Y_{i+1},\ldots,Y_k\} \subseteq \texttt{nullable} &\Rightarrow \texttt{follow}(X) \subseteq \texttt{follow}(Y_i) \\
\{Y_{i+1},\ldots,Y_{j-1}\} \subseteq \texttt{nullable} &\Rightarrow \texttt{first}(Y_j) \subseteq \texttt{follow}(Y_i)
\end{aligned}$$

On peut réécrire cette définition comme suit :

$$\begin{aligned}
\{Y_1,\ldots,Y_k\} \subseteq \texttt{nullable} &\Rightarrow X \in \texttt{nullable} \\
\{Y_1,\ldots,Y_{i-1}\} \subseteq \texttt{nullable} &\Rightarrow \texttt{first}(X) = \texttt{first}(X) \cup \texttt{first}(Y_i) \\
\{Y_{i+1},\ldots,Y_k\} \subseteq \texttt{nullable} &\Rightarrow \texttt{follow}(Y_i) = \texttt{follow}(Y_i) \cup \texttt{follow}(X) \\
\{Y_{i+1},\ldots,Y_{j-1}\} \subseteq \texttt{nullable} &\Rightarrow \texttt{follow}(Y_i) = \texttt{follow}(Y_i) \cup \texttt{first}(Y_j)
\end{aligned}$$

Pour calculer `first(X)`, `follow(X)` et `nullable`, on utilise une itération aux points fixes basées sur cette dernière définition.

Algorithme : `first(X)`, `follow(X)` et `nullable`

```
for (val term <- terminals) first(term) = first(term) ∪ term
var fixpointReached = false
do {
  val nullableCheck = nullable; val firstCheck = first; val followCheck = follow
  for (val Production(left, right) <- productions) {
    val k = right.length + 1
    if (right ⊆ nullable) nullable = nullable ∪ left
    for (val i <- List.range(1, k))
      if (right_1 ... right_{i-1}) ⊆ nullable)
        first(left) = first(left) ∪ first(right_i))
    for (val i <- List.range(1, k))
      if (right_{i+1} ... right_k) ⊆ nullable)
        follow(right_i) = follow(right_i) ∪ follow(left)
    for (val i <- List.range(1, k-1); val j <- List.range(i+1, k))
      if (right_{i+1} ... right_{j-1}) ⊆ nullable)
        follow(right_i) = follow(right_j) ∪ first(right_j)
  }
  fixpointReached = nullableCheck == nullable && firstCheck == first && followCheck == follow
} while (!fixpointReached);
```

Sachant qu'une production $X = Y_1 \ \ldots \ Y_n$ sera représentée par `Production(X, List(`Y_1`, ..., `Y_n`))`.

Une implémentation de cet algorithme est disponible sur le site.

De l'EBNF à la BNF simple

Notez que l'algorithme précédent requiert une grammaire sous forme BNF simple. Ceci est facile à obtenir à partir d'une grammaire EBNF :

- changer chaque répétition `{E}` dans la grammaire en un nouveau symbole non-terminal X_{rep} et ajouter la production X_{rep} `= E` X_{rep} `| (empty).`.
- changer chaque option `[E]` dans la grammaire en un nouveau symbole non-terminal X_{opt} et ajouter la production X_{opt} `= E | (empty).`.
- supprimer les alternatives en les remplaÁant par plusieurs productions pour le même symbole non-terminal : `X = E X | (empty).` devient `X = E X. X = (empty).`

Grammaires LL(1)

Définition : Grammaire LL(1)

Une grammaire BNF simple est LL(1) si pour tout non-terminal `X` apparaissant dans le membre de gauche de deux productions : `X = E1. X = E2.`

Alors :

1. `first(E1)` $\cap$ `first(E2)` $= \{\}$.
2. Une des conditions suivantes est vrai :
   - ni `E1` ni `E2` n'est annulable ;
   - exactement un `E` est annulable et `first(X)` $\cap$ `follow(X)` $= \{\}$.

LL(1) signifie ´analyse de gauche à droite (*left-to-right parse*), dérivation la plus à gauche (*leftmost derivation*), 1 symbole lu en avance (*1 symbol lookahead*)[a].

Convertir vers LL(1)

Les analyseurs syntaxiques par descente récursive marchent seulement pour les grammaires LL(1).

Exercice

La grammaire pour les expressions arithmétiques sur les tableaux :

```
E  =  E "+" T | E "-" T | T.
T  =  T "*" F | T "/" F | F.
F  =  ident | ident "[" E "]" | "(" E ")".
```

est-elle LL(1) ?

Pour convertir une grammaire vers LL(1) :

- on élimine la récursivité à gauche ;

Exemple : Conversion en LL(1)

```
E  =  E "+" T | E "-" T | T.
```

devient :

```
E  =  T { "+" T | "-" T}.
```

- on factorise à gauche.

```
F  =  ident | ident "[" E "]" ... .
```

devient :

```
F  =  ident ( (empty) | "[" E "]") ... .
```

ou bien, en utilisant une option :

```
F  =  ident [ "[" E "]" ] ... .
```


L'élimination de la récursivité à gauche et la factorisation à gauche marchent souvent, mais pas toujours.

Exemple

```
S  =  { A }.
A  =  ident ":=" E.
E  =  { ident }.
```

On ne peut pas donner de grammaire LL(1) à ce langage. Mais il est LL(2), c'est-à-dire qu'il peut être analysé avec 2 symboles lus en avance.

- De faÁon générale LL(k) est un sous-ensemble strict de LL(k+1).
- Mais LL(1) est le seul cas intéressant en pratique.

Résumé : Analyse syntaxique descendante

D'une grammaire non-contextuelle on peut extraire directement un schéma de programmation pour un analyseur syntaxique par descente récursive.

Un analyseur syntaxique par descente récursive construit une dérivation du haut vers le bas (*top-down*), du symbole initial vers les symboles terminaux.

Mais attention :

- il doit décider quoi faire en se basant sur le premier symbole d'entrée ;
- et cela ne marche que si la grammaire est LL(1).

Analyse syntaxique ascendante

Un analyseur syntaxique ascendant (*bottom-up*) crée une dérivation à partir des symboles terminaux, en remontant vers le symbole initial.

- Il consiste en une pile (*stack*) et une entrée (*input*).
- Ses quatre actions sont :
  1. décaler (*shift*) : empile le prochain symbole d'entrée ;
  2. réduire (*reduce*) : retire les symboles $Y_n, \ldots, Y_1$ qui forment le membre droit d'une production `X = Y1 ... Yn` ;
  3. accepter ;
  4. erreur.

Exercice

Comment l'analyseur syntaxique sait-il quand décaler et quand réduire ?

Priorité des opérateurs

Une réponse simple à cette question est la priorité des opérateurs.

- Celle-ci est adapteé aux langages de la forme :
  `Expression = Opérande Opérateur Opérande`
  avec des opérandes de priorité et associativité variable.

Algorithme

`IN` ← prochain symbole d'entrée ;
si `IN` est une opérande → décaler,
sinon si la pile ne contient pas un opérateur → décaler,
sinon :
  `TOP` ← l'opérateur le plus haut sur la pile ;
  si la priorité de `TOP` < la priorité de `IN` → décaler,
  sinon si la priorité de `TOP` > la priorité de `IN` → réduire,
  sinon si `IN` et `TOP` sont associatifs à droite → décaler,
  sinon si `IN` et `TOP` sont associatifs à gauche → réduire,
  sinon erreur.

Analyse LR(0)

Une réponse plus générale à cette question est l'analyse LR(0) où un AFD appliquée à la pile décide quand décaler et quand réduire.

Les états de l'AFD sont des ensembles d'éléments LR(0) :

- Un élément LR(0) a la forme `[ X = A . B ]`
  où `X` est un symbole non-terminal et `A`, `B` sont des chaînes de symboles éventuellement vides.
- Un élément LR(0) décrit une situation possible pendant l'analyse où :
  - `X = AB` est une production possible pour la dérivation courante ;
  - `A` est sur la pile ;
  - `B` reste sur l'entrée.
- Donc, le `.` décrit la frontière entre la pile et l'entrée.

Le principe de fonctionnement est le suivant :

- Décaler dans un état qui contient l'élément `[X = A . b B]` si le prochain symbole d'entrée est `b`.
- Réduire dans un état qui contient l'élément `[X = A .]`.

Vous trouverez des examples dans l'*Appel*, figure 3.20.

L'analyseur obtenu est appelé LR(0) car il analyse l'entrée de gauche à droite (*left-to-right*) et décrit une dérivation la plus à droite (*right-most*) à l'envers. Le 0 signifie que l'analyseur ne lit aucun symbole en avance.

Analyse SLR

Problème : certains états contiennent à la fois des éléments décaler et réduire.

Exemple

```
S  =  E "=".          T  =  "(" E ")".
E  =  T "+" E.        T  =  "x".
E  =  T.
```

La construction des états LR(0) donne un état contenant les éléments `[ E = T. + E ]` et `[ E = T. ]`.

Si le prochain symbole d'entrée est `+`, doit-on décaler ou réduire ?

Solution : réduire seulement si le prochain symbole est dans `follow(E)`.

L'analyseur obtenu est appelé *´simple LR*[a], ou SLR.

Analyse LR(1)

L'analyse LR(1) est plus puissante que l'analyse SLR. Elle raffine encore la notion d'état.

Un état est maintenant un ensemble d'éléments LR(1) de la forme `[X = A . B ; c]` où :

- `X = AB` est une production de la grammaire,
- `A` est sur la pile et `Bc` fait partie de l'entrée.

Le reste de la construction est similaire à LR(0), sauf que l'on réduit dans un état avec l'élément `[X = A . ; c]` seulement si le prochain symbole d'entrée est `c`.

Le résultat est appelé analyse LR(1), car elle lit les entrées de gauche à droite (*left-to-right*), décrit une dérivation la plus à droite (*right-most*) à l'envers, et utilise 1 symbole lu en avance.

Analyse LALR(1)

Problème : il y a beaucoup plus d'états LR(1) que d'états LR(0) : l'explosion des états pose problème.

Solution : fusionner les états qui ne diffèrent que par le symbole lu en avance (*lookahead symbol*).

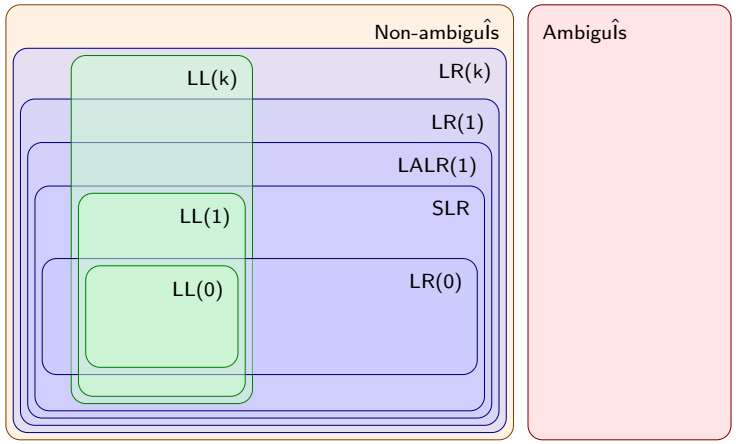
Exemple

Les deux états `{[X = A . B ; c]}, {[X = A . B ; d]}` deviennent `{[X = A . B ; c], [X = A . B ; d]}`

L'analyseur obtenu est appelé LALR(1) pour *Look-Ahead-LR(1)*.

La technique LALR(1) est à la base de la plupart des générateurs de analyseurs comme *Yacc*, *Bison*, *CUP*.

Hiérarchie des classes de grammaires



Exemple : Spécification d'un analyseur syntaxique

```
terminal          ID, WHILE, BEGIN, END, DO, IF, THEN,
                  ELSE, SEMI, ASSIGN;
non terminal      prog, stm, stmlist;
start             prog;

prog          ::= stmlist;
stm           ::= ID ASSIGN ID
                | WHILE ID DO stm
                | BEGIN stmlist END
                | IF ID THEN stm
                | IF ID THEN stmELSE stm;
stmlist       ::= stm
                | stmlist SEMI stm;
```


Pragmatisme

La grammaire de Java n'est pas LL(1) ou LR(1) ; elle n'est même pas non-ambiguÎ !

Exemple : Syntaxe ambiguÎ de Java

```
    Block   =  { Statement }.
Statement   =  "if" "(" Expression ")" Statement
               [ "else" Statement ].
           |   Assignment.
```

Comment analyse-t-on le code suivant ?

```
if (x != 0) if (x < 0) y = -1 else y = 1
```


Exercice

Réécrire la grammaire pour qu'elle devienne non-ambiguÎ.

Solutions pragmatiques :

- Descente récursive : appliquer la règle de la plus longue correspondance (*longest match rule*).
- LR(x) : avoir des priorités sur les règles. C'est-à-dire que les règles les plus anciennes ont priorité sur les plus récentes :

```
Statement   =  "if" "(" Expression ")" Statement
               [ "else" Statement ].
Statement   =  "if" "(" Expression ")" Statement.
```

Résumé : Approche ascendante contre descendante

Descendante

\+
Facile à écrire à la main
Intégration flexible au compilateur
–
Plus difficile à maintenir
Traitement des erreurs mal-aisé
Récursivité peut être inefficace

Ascendante

\+
Facile à maintenir
Plus large classe de langages
–
Requiert des outils de génération
Intégration rigide au compilateur
Dépend de la qualité de l'outil

Des mélanges sont possibles. Beaucoup d'analyseurs syntaxiques de compilateurs commerciaux utilisent la descente récursive avec des priorités sur les opérateurs pour les expressions pour se débarrasser de la récursivité profonde.

Diagnostiques d'erreur

Quand il rencontre un programme d'entrée illégal, l'analyseur doit afficher un message d'erreur.

Exercice

Quel message d'erreur doit-on afficher pour :

```
x[i) = 1; ou
x = if (a < b) 1 else2;
```

Cela aide souvent d'inclure l'entrée qui a effectivement été rencontrée :

```
"{" expected but identifier found
```

On peut utiliser la fonction `representation` dans l'analyseur syntaxique pour ce travail.

Reprise après erreur

Après une erreur, l'analyseur doit être capable de continuer le traitement.

Dans ce cas, le traitement a pour but de trouver d'autres erreurs, pas de générer du code : La génération de code doit être désactivée.

Mais comment l'analyseur récupère-t-il d'une erreur et reprend-il le traitement normal ?

Deux éléments de solution :

- Sauter une partie de l'entrée.
- Réinitialiser l'analyseur dans un état où il peut traiter le reste de l'entrée.

Reprise après erreur (descente récursive)

Soit $X_1 \ldots X_n$ la pile des méthodes d'analyse (correspondantes chacune à une production) qui sont en train de s'exécuter.

Pour un i quelconque :

- sauter l'entrée jusqu'à un symbole dans `follow(`X_i`)` ;
- réduire la pile des méthode jusqu'au point de retour de X_i.

En pratique on peut se contenter d'une solution plus simple pour sauter l'entrée :

- on détermine un ensemble fixe de symboles d'arrêt qui terminent un saut tels que `;`, `}`, `)` ou `EOF`.
- on s'assure de sauter entièrement un sous-bloc en comptant les parenthèses et accolades ouvrantes.

Exemple : Sauter des sous-blocs

```
if x < 0 { ... }
   ^ '(' expected but identifier found
```

ne doit pas sauter jusqu'à `}`.

Algorithme : Sauter l'entrée en comptant des parenthèses

```
def skip: Unit = {
  var nbParens = 0
  while (true) {
    token match {
      case EOF => return
      case SEMI => if (nbParens == 0) return
      case LPAREN | LBRACE => nbParens = nbParens + 1
      case RPAREN | RBRACE =>
        nbParens = nbParens - 1; if (nbParens == 0) return
      case _ => /* Nothing to do */
    }
    nextToken
  }}
```


Réduire la pile des méthodes

Problème : réduire la pile des méthodes explicitement pour la reprise après erreur de l'analyseur n'est pas faisable en Scala.

Solution : c'est en fait très simple, il suffit de continuer comme si rien ne s'était passé !

- L'analyseur va finir par atteindre un état où il pourra accepter sur l'entrée le symbole d'arrêt qui suit directement un saut ; il va se re-synchroniser.
- Attention : pour que l'analyseur termine, il doit invoquer une méthode `X()` que si le prochain symbole d'entrée est dans `first(X)`.

Récupérer les erreurs de syntaxe

Problème : la solution de gestion des erreur esquissée précédemment va générer beaucoup de faux messages d'erreurs avant que l'analyseur ne se re-synchronise.

Solution : il suffit de ne pas afficher de messages d'erreur tant que l'analyseur n'a pas consommé au moins un autre symbole d'entrée.

Algorithme : Cacher des messages d'erreur superflus

Soit `pos` la position du prochain lexème d'entrée et `skipPos` la dernière position que l'on a sauté :

```
def syntaxError (message: String): Unit {
  if (pos != skipPos) errorMsg(pos, message)
  skip
  skipPos = pos
}
```

Reprise après erreur (analyse ascendante)

De nombreux schémas sont possible pour la résolution d'erreurs dans l'analyse ascendante.

Dans Yacc, Bison et CUP, la solution se base sur l'introduction d'un symbole spécial `error` :

L'auteur de la grammaire syntaxique peut ainsi utiliser `error` dans les productions.

Exemple : Ajouter explicitement les erreurs à la grammaire

```
Block   =   "{" {Statement} "}"
        |   "{" {Statement} error "}".
```


Si l'analyseur rencontre une erreur, il va réduire la pile jusqu'à ce qu'il atteigne un état où `error` est un prochain symbole légal.

Ensuite, il saute les symboles d'entrée jusqu'à ce que le prochain symbole d'entrée puisse légalement suivre dans le nouvel état.

Exemple : Décaler `error`

A partir de cet état de l'analyseur :

```
[ Block = "{" {Statement} . error "}" ]
```

on pourra décaler `error` comme ceci :

```
[ Block = "{" {Statement} error . "}" ]
```

et *re-synchroniser* l'analyse au prochain symbole `}`.

Ce schéma est très dépendent d'un bon choix dans les productions d'erreur.