Partie III : Analyse syntaxique

- Des grammaires régulières aux grammaires non contextuelles
- Dériver un analyseur syntaxique d'une grammaire non contextuelle
- Analyseurs lexicaux et analyseurs syntaxiques
- Un analyseur syntaxique pour la notation EBNF
- Grammaires analysables par la gauche (*left-parsable*)

Des grammaires régulières aux grammaires non contextuelles

- Les langages réguliers sont limités : ils ne peuvent pas exprimer l'imbrication.
- Donc les machines à états finis ne peuvent pas reconnaître les grammaires non contextuelles.
- Mais essayons quand même !
- **Exemple** : La grammaire :

```
A = "a" A "c" | "b".
```

conduit après simplification au "reconnaisseur" suivant :

Des grammaires régulières aux grammaires non contextuelles (2)

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

- C'est incorrect, bien entendu, car nous avons traité le symbole non-terminal A comme un symbole terminal.
- Mais cela suggère une extension naturelle de notre algorithme de reconnaissance.

Dériver un analyseur syntaxique d'une grammaire non contextuelle

- Pour dériver un analyseur syntaxique d'une grammaire non contextuelle écrite dans le style EBNF :
- Introduire une fonction `void A()` pour chaque non-terminal `A`.
- Le travail de `A()` est de reconnaître les sous-phrases dérivées de `A`, ou d'émettre une erreur si aucun `A` n'a été trouvé.
- Traduire toutes les expressions régulières des membres droits d'une production comme avant, avec en plus que :
  - chaque occurrence d'un non-terminal `B` donne `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 syntactique par *descente récursive* ou *analyse prédictive*.

Dériver un analyseur syntaxique d'une grammaire non contextuelle (2)

Exemple : La grammaire :

```
A = "a" A "c" | "b".
```

conduit maintenant à la fonction :

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


Analyseurs lexicaux et syntaxiques

- En pratique la plupart des compilateurs ont à la fois un analyseur lexical pour la syntaxe lexicale et un analyseur syntaxique pour la syntaxe non contextuelle.
- Avantages : séparation des objectifs, meilleure modularité.

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

Un analyseur syntaxique pour la notation EBNF

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

```
package ebnf;
import java.io.*;
class Parser extends Scanner {
    public Parser(InputStream in) {
        super(in);
        nextSym();
    }
```

- Maintenant, traduisons simplement chaque production selon le schéma suivant.
- Voilà la production pour analyser toute une grammaire :

```
/* syntax = {production} <eof> */
public void syntax() {
   while (sym != EOF) {
     production();
   }
}
```


Un analyseur syntaxique pour la notation EBNF (2)

Et voici les productions pour les autres non-terminaux :

```
/* production = identifier "=" expression "." */
void production() {
    if (sym == IDENT) nextSym();
    else error("illegal start of production");
    if (sym == EQL) nextSym();
    else error("`=' expected");
    expression();
    if (sym == PERIOD) nextSym();
    else error("`.' expected");
}
/* expression = term {"|" term} */
void expression() {
    term();
    while (sym == BAR) {
        nextSym();
        term();
    }
}
```

Un analyseur syntaxique pour la notation EBNF (3)

```
/* 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:
          nextSym();
          break;
     case LITERAL:
          nextSym();
          break;
```


Un analyseur syntaxique pour la notation EBNF (4)

```
    case LPAREN:
        nextSym();
        expression();
        if (sym == RPAREN) nextSym();
        else error("`)' expected");
        break;
    case LBRACK:
        nextSym();
        expression();
        if (sym == RBRACK) nextSym();
        else error("`]' expected");
        break;
    case LBRACE:
        nextSym();
        expression();
        if (sym == RBRACE) nextSym();
        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.

Deux conditions :

- Dans une alternative $T_1 \mid \ldots \mid T_n$, les termes ne doivent pas avoir de symbole de départ commun.
- 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.

De l'EBNF à la BNF simple

Il est facile de transformer une grammaire EBNF en BNF :

- changer chaque répétition {E} dans une grammaire EBNF en un nouveau symbole non-terminal X_{rep} et ajouter la production :

  $X_{rep} = E\ X_{rep} \mid \epsilon$

- changer chaque option [E] dans la grammaire en un nouveau symbole non-terminal X_{opt} et ajouter la production :

  $X_{opt} = E \mid \epsilon$

- On peut même se débarrasser des alternatives en ayant plusieurs productions pour le même symbole non-terminal. Exemple :

  $X_{rep} = E\ X_{rep} \mid \epsilon$

devient

$X_{rep} = E\ X_{rep}$
$X_{rep} = \epsilon$

Définition de first, follow et nullable

étant donné un langage non contextuel, 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-à-d, t ∈ follow(X) s'il y a une dérivation qui contient X t.

Définition formelle de first, follow et nullable

first, follow et nullable sont les plus petits ensembles qui vérifient ces propriétés.

for each production $X = Y_1 \ldots Y_k$, $1 \le i < j \le k$:
- if $\{ Y_1, \ldots, Y_k \} \subseteq$ nullable
  - $X \in$ nullable
- if $\{ Y_1, \ldots Y_{i-1} \} \subseteq$ nullable
  - first(X) = first(X) ∪ first(Y_i)
- if $\{ Y_{i+1}, \ldots, Y_k \} \subseteq$ nullable
  - follow(Y_i) = follow(Y_i) ∪ follow(X)
- if $\{ Y_{i+1}, \ldots, Y_{j-1} \} \subseteq$ nullable
  - follow(Y_i) = follow(Y_i) ∪ first(Y_j)

Algorithme pour calculer first, follow et nullable

Remplacer simplement les équations par des affectations et itérer jusqu'à ce que plus rien ne change.

```
nullable := {}
for each terminal t: first( t ) := {t} ; follow( t) := {}
for each nonterminal Y: first( Y ) := {} ; follow( Y ) := {}
repeat
  nullable' := nullable ; first' := first ; follow' := follow
  for each production X = Y1 ... Yk, 1 ≤ i < j ≤ k:
    if { Y1, ..., Yk }  nullable
      nullable := nullable ∪ { X }
    if { Y1, ... Yi-1 } ⊆ nullable
      first( X ) := first( X ) ∪ first( Yi )
    if { Yj+1, ..., Yk } ⊆ nullable
      follow( Yi ) := follow( Yi ) ∪ follow( X )
    if { Yi+1, ..., Yj-1 } ⊆ nullable
      follow( Yi ) := Follow( Yi ) ∪ first( Yj )
until nullable' = nullable & first' = first & follow' = follow.
```


Grammaires LL(1)

- Définition : une grammaire BNF simple est LL(1) si pour tout non-terminal X:
  si X apparaît dans le membre gauche de deux productions

  X = E1

  X = E2

  alors

  first(E1) ∩ first(E2) = {}

  et soit

  ni E1, ni E2 n'est annulable

  soit

  exactement un E_i est annulable et
  first(X) ∩ 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*) ".
- Les analyseurs syntaxiques par descente récursive marchent seulement pour les grammaires LL(1).

Convertir en LL(1)

- Exemple : 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 ")"

- Cette grammaire est-elle LL(1) ?

Techniques :

- Éliminer la récursivité à gauche. E.g.

  E = E "+" T | E "-" T | T

  devient

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

- Factorisation à gauche: E.g.

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

  devient

  F = ident (∈ | "[" E "]") | ...

  ou bien, en utilisant une option:

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

Limitations

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

  S = { A }.
  A = id ":=" E.
  E = {id}.

- On ne peut pas donner de grammaire LL(1) à ce langage. Mais il est LL(2), i.e. il peut être analysé avec 2 symboles lus en avance.
- De façon 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.
- Faiblesse : il doit décider quoi faire en se basant sur le premier symbole d'entrée.
- 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*).
- Les deux actions de base :

  décaler (*shift*) : empiler le prochain symbole d'entrée

  réduire (*reduce*) : retirer les symboles $Y_n,\ldots,Y_1$ qui forment le membre droit d'une production

  $X = Y_1 \ldots Y_n$

  du sommet de la pile et les remplacer par X.
- Autres actions : accepter, erreur.
- Question : Comment l'analyseur syntaxique sait-il quand décaler et quand réduire ?

Réponse simple : Priorité des opérateurs

- Adapté aux langages de la forme

  Expression = Operand Operator Operand.

  avec des opérandes de priorité et associativité variable.
- Principe:

```
soit IN le prochain symbole d'entrée.
si IN est une opérande alors
   décaler
sinon si la pile ne contient pas un opérateur alors
   décaler
sinon
  soit TOP l'opérateur le plus haut sur la pile.
  si priorité(TOP) < priorité(IN) alors décaler
  sinon si priorité(TOP) > priorité(IN) alors réduire
  sinon si IN et TOP sont associatifs à droite alors décaler
  sinon si IN et TOP sont associatifs à gauche alors réduire
  sinon erreur
```

Réponse plus générale : analyse LR(0)

- Idée : Utiliser un AFD appliquée à la *pile* pour décider 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.

Réponse plus générale : analyse LR(0) (2)

- Principe:
  - 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 .]
- Exemple: Voir 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 : Considérons la grammaire :

  S = E $
  E = T + E
  E = T
  T = (E)
  T = x

- La construction des états LR(0) donne un état contenant les éléments

  E = T. + E
  E = T.

- Si on voit "+" comme prochain symbole d'entrée, doit-on décaler ou réduire ?
- Solution : Réduire seulement si le prochain symbole est dans follow(E).
- L'analyseur obtenu est appelé "simple LR", ou *SLR*.

Analyse LR(1)

- L'analyse LR(1) est encore plus puissante que l'analyse SLR.
- L'analyse LR(1) raffine la notion d'état. Un état est maintenant un ensemble d'éléments LR(1), où chaque élément a la forme

  [X = A . B ; c]

- Cela modélise la situation suivante

  X = AB est une production de la grammaire
  A est sur la pile
  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 (lookahead symbol).

Analyse LALR(1)

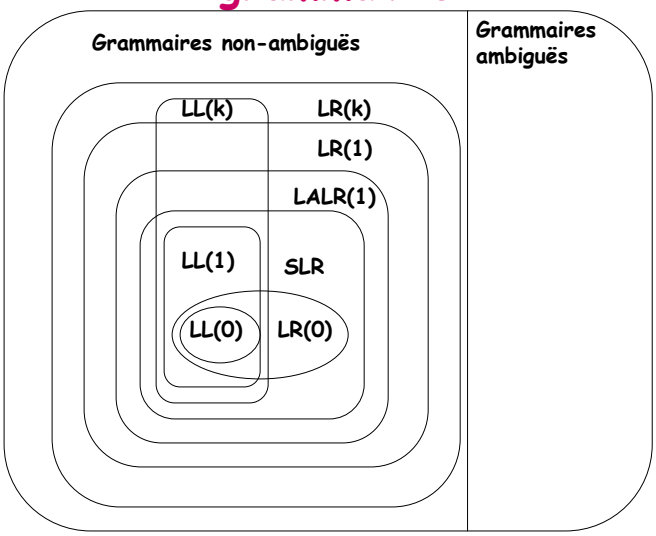
- Les analyseurs LR(1) sont plus puissants que les analyseurs SLR.
- Mais : Il y a beaucoup plus d'états LR(1) que d'états LR(0). Souvent, *l'explosion des états* pose problème.
- Solution : Fusionner les états qui ne diffèrent que par le symbole lu en avance (*lookahead symbol*).
- Exemple : Les deux 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, p.ex. Yacc, Bison, JavaCUP.

Une hiérarchie des classes de grammaires



Exemple de spécification d'un analyseur syntaxique

```
terminal      ID, WHILE, BEGIN, END, DO, IF, THEN,
              ELSE, SEMI, ASSIGN;

non terminal  prog, stm, stmlist;

start with    prog;

prog      ::= stmlist;

stm       ::= ID ASSIGN ID
          |   WHILE ID DO stm
          |   BEGIN stmlist END
          |   IF ID THEN stm
          |   IF ID THEN stm ELSE stm;

stmlist   ::= stm
          |   stmlist SEMI stm;
```


Pragmatisme

- La grammaire de Java est-elle LL(1) ou LR(1) ?
- Elle n'est même pas non-ambiguë !
- Problème :

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

- Comment analyse-t-on

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

- ?

Pragmatisme (2)

- Exercice : Réécrire la grammaire pour qu'elle devienne non-ambiguë.
- Solutions pragmatiques :
  - Descente récursive : appliquer la règle de la plus longue correspondance (*longest match rule*)
  - LR : Avoir des priorités sur les règles. E.g., les règles les plus anciennes ayant priorité sur les plus récentes :

```
Statement = "if" "(" Expression ")" Statement "else"
            Statement
Statement = "if" "(" Expression ")" Statement
```


Avantages relatifs des approches ascendantes et descendantes :

Descendante : + Facile à écrire à la main
+ Intégration flexible dans un compilateur
- Plus difficile à maintenir
- Le traitement des erreurs peut être mal-aisé
- La récursivité profonde peut être inefficace.

Ascendante : + Classe plus large de langages et grammaires
+ Maintenabilité
- nécessite des outils de génération
- Moins flexible pour intégrer dans un 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.
- Question: quel message d'erreur doit-on afficher pour :

```
x [i) = 1;
```

- Et pour :

```
x = if (a < b) 1 else 2;
```

- Cela aide souvent d'inclure l'entrée qui a effectivement été rencontrée. E.g.

  "{" expected but identifier found

- On peut utiliser la fonction `representation` dans le scanner pour ce travail.

Reprise après erreur

- Après une erreur, l'analyseur doit être capable de continuer le traitement.
- Le traitement a pour but de trouver d'autres erreurs, pas de générer du code.

  ⇒ La génération de code doit être désactivée.

- Question : 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 lors d'une descente récursive

- Soit $X_1 \ldots X_n$ la pile courante des méthodes d'analyse qui sont en train de s'exécuter.
- Idée : Sauter l'entrée jusqu'à un symbole dans follow(X_i) pour un i et dépiler la pile jusqu'au point de retour de X_i.
- En pratique, il est souvent suffisant d'avoir un ensemble fixe de symboles d'arrêt qui terminent un saut. E.g., pour FuLL :

  ";", "}", ")", EOF
- En pratique, il est aussi important de sauter entièrement les sous-blocs.

  Exemple:

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

  Ne doit pas sauter jusqu'à "}" !
- Cela peut être réalisé en comptant les parenthèses et accolades ouvrantes.

Une procédure pour "passer"

```
void skip () {
          int nparens = 0;
          while (true) {
            switch (token) {
            case EOF    : return;
            case SEMI   : if (nparens == 0) return;
                          break;
            case LPAREN:
            case LBRACE: nparens++; break;
            case RPAREN:
            case RBRACE: if (nparens == 0) return;
                          nparens--; break;
             }
             nextToken();
          }
       }
```

Dépiler la pile

- Problème : Comment dépile-t-on la pile dans un analyseur à descente récursive ?
- Une solution étonnamment simple : continuer simplement l'analyse, comme si rien ne s'était passé !
- L'analyseur va finir par atteindre un état où il pourra accepter le symbole d'arrêt qui suit directement un saut sur l'entrée ; il va se *resynchroniser*.
- Pré-requis pour la terminaison : l'analyseur ne doit invoquer une méthode X() que si le prochain symbole d'entrée est dans first(X).
- Mais cela va générer beaucoup de faux messages d'erreurs avant que l'analyseur ne se resynchronise !
- Solution : Après un saut, ne pas afficher de messages d'erreur tant que l'analyseur n'a pas consommé au moins un autre symbole d'entrée.

Récupérer les erreurs de syntaxe

- Introduire une variable globale `pos` pour la position du prochain lexème d'entrée (soit en lignes et colonnes, soit en nombre de caractères depuis le début).
- `pos` doit être maintenu par le scanner.
- Introduire une variable globale `skipPos` pour la dernière position que l'on a sauté.

```
int skipPos = -1
```

- Définir maintenant une procédure pour récupérer les erreurs de syntaxe comme suit.

```
/** Signaler une erreur de syntaxe à moins qu'une
    autre ait déjà été signalée à cette position
    puis sauter. */
private void syntaxError(String msg) {
    if (pos != skipPos) errorMsg(pos, msg);
    skip();
    skipPos = pos;
}
```

- Cela est très simple et marche bien en pratique.

Reprise après erreur avec l'analyse ascendante

- De nombreux schémas sont possible. Voici celui implémenté dans Yacc, Bison et JavaCUP:
- Introduire un symbole spécial `error`.
- L'auteur de l'analyseur syntaxique peut utiliser `error` dans les productions.
- Par exemple :

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


Reprise après erreur avec l'analyse ascendante (2)

- 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.

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

- A ce moment-là, on décalera `error` :

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

- Ensuite, on saute les symboles d'entrée jusqu'à ce que le prochain symbole d'entrée puisse légalement suivre dans le nouvel état.
- Ce schéma est très dépendent d'un bon choix dans les productions d'erreur.