Au menu de l'UE Compilation

Introduction

I - Programmation dirigée par la syntaxe

II - Analyse lexicale - Analyse syntaxique

III – Description formelle d'un langage

III.1 – Expressions régulières (ou rationnelles) – Automates finis

III.2 – Grammaires algébriques (ou hors-contexte)

A - Programmation par automates finis

I - Automates finis avec actions

I.1 – Analyse syntaxique et actions

I.2 – Analyse lexicale et actions

II - Programmation d'un automate fini déterministe

II.1 – Programmation directe

II.2 – Programmation par table

II.3 – Traitement des erreurs

B - Analyse syntaxique descendante de gauche à droite (DGD)

I - Limite des automates finis

II - Analyseur DGD procédural - Points de génération

C – Construction d'un compilateur

I - Compilateur

II - Table des symboles - Compilation des déclarations

III - Compilation des expressions - Calcul de type

IV - Compilation des instructions

V - Compilation des procédures

VI - Compilation séparée - Édition de liens

D - Automates à pile - Grammaires LL(1)

I – Analyse DGD par automate à pile

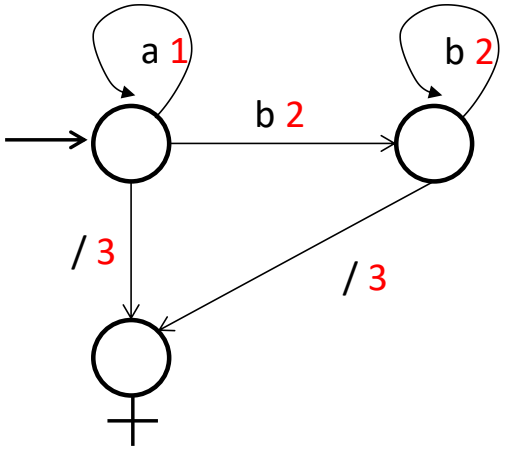
II - Analyse DGD et grammaire LL(1)

III – Analyseur associé à une grammaire LL(1)

B - Analyse syntaxique DGD
I - Limite des automates finis

Exemple 1 - Cas particulier

- Le langage L = {$a^n b^n$ /} n'est pas rationnel. On peut cependant l'analyser à l'aide d'un automate fini augmenté d'actions avec un compteur.
- on exhibe un automate reconnaissant un sur-langage reconnaissable de L = {$a^m b^p$ /} : a*b* /
- dans les actions, on compte les a et les b pour contrôler l'égalité.




```
0 : cpt = 0;
1 : cpt = cpt+1;
2 : cpt = cpt-1;
3 : if (cpt !=0)
    erreur(FATALE, "erreur de syntaxe");
```

B - Analyse syntaxique DGD
I - Limite des automates finis

Exemple 2 - Cas général

Plus compliqué dès que le langage d'entrée comporte **plusieurs catégories de « parenthésages » pouvant s'imbriquer**

Cas des langages de programmation
comme Java : if … {…}, while … {…}, switch …{…}
comme langage PROJET : si … fsi, faire … fait, cond … fcond (…)
Ex: si si cond faire <Instructions> => « fermeture » dans l'ordre : fait fcond fsi fsi

- Un simple compteur entier n'est pas adapté pour une imbrication
- nécessité d'une **pile**
- Au lieu de gérer une pile au niveau des actions d'un automate fini, on utilise un **automate à pile** pour l'analyse syntaxique du langage d'entrée (cf. LF).

Au menu de l'UE Compilation

B - Analyse syntaxique DGD

II - Analyseur DGD procédural - Points de génération

Rappels grammaire algébrique

- Une grammaire G est un quadruplet (V_T, V_N, S, P) où :
  - V_N alphabet non-terminal disjoint de V_T
  - $S \in V_N$ est l'axiome
  - P est l'ensemble fini des règles de production

- les règles étant de la forme :
  - $X \rightarrow a_1 \mid a_2 \mid \ldots \mid a_n$ avec n règles, $n \geq 1$
  - avec $X \in V_N$ et $a_i \in (V_T \cup V_N)^*$

B - Analyse syntaxique DGD

II - Analyseur DGD procédural - Grammaires algébriques

Remarques

- Grammaires algébriques employées pour décrire
  - des langages rationnels (ex : langage Monnaie)
  - des langages algébriques (non rationnel) et donc non analysables par automate fini.
- La machine adaptée est alors l'***automate à pile***.

- Soit une grammaire G (S axiome), le langage engendré par G est $L(G) = \{ w \in V_T^* \mid S \rightarrow^+ w \}$
- Un mot du langage peut être représenté par un ***arbre syntaxique*** (***arbre de dérivation***).

- Écriture EBNF des règles de grammaire (notations régulières en partie droite des règles) :
  - <suite_cotations> → (<cotation> PTVIRG)* BARRE

B - Analyse syntaxique DGD

II - Analyseur DGD procédural - Points de génération

Analyse des langages algébriques

- Utilisation d'un automate à pile (les détails + tard)
- Un tel automate permet
  - Une Analyse Descendante de Gauche à Droite (DGD) du langage L(G) engendré par une grammaire algébrique G
  - **N.B**. La grammaire doit être non-ambiguë et respecter d'autres contraintes (vues plus tard)

- **Principes**
  - À partir de l'axiome, ***dérivation gauche***, pour obtenir la chaîne analysée.
  - À chaque pas, le non-terminal le plus à gauche est dérivé.

B - Analyse syntaxique DGD

II - Analyseur DGD procédural - Points de génération

- **Exercice 1 :** Soit la grammaire algébrique définie sur V_T = { a, b, c, d }
  - S → a Y Z | c
  - Y → a Y | d | ε
  - Z → b

Construire une dérivation gauche (si elle existe) puis **dessiner** l'arbre syntaxique correspondant, pour les chaînes suivantes

1. aadb
2. aab
3. ac

B - Analyse syntaxique DGD

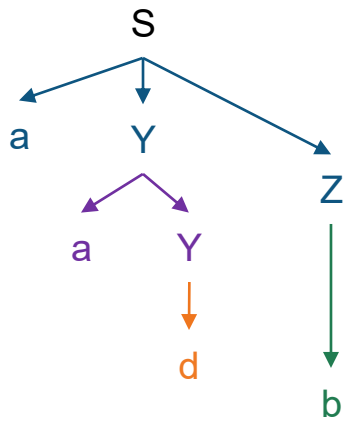
II - Analyseur DGD procédural - Points de génération

- **Exercice 1 :** Soit la grammaire algébrique définie sur V_T = { a, b, c, d }
  - S → a Y Z | c
  - Y → a Y | d | ε
  - Z → b

aadb

- **Dérivation gauche**
  - **S**

- **Arbre syntaxique**



B - Analyse syntaxique DGD

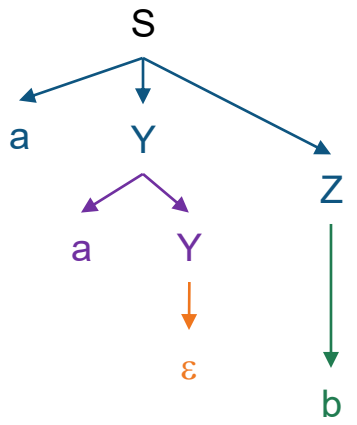
II - Analyseur DGD procédural - Points de génération

- **Exercice 1 :** Soit la grammaire algébrique définie sur V_T = { a, b, c, d }
  - S → a Y Z | c
  - Y → a Y | d | ε
  - Z → b

aab

- **Dérivation gauche**
  - **S**

- **Arbre syntaxique**



B - Analyse syntaxique DGD

II - Analyseur DGD procédural - Points de génération

- **Exercice 1 :** Soit la grammaire algébrique définie sur V_T = { a, b, c, d }
  - S → a Y Z | c
  - Y → a Y | d | ε
  - Z → b

ac

- **Dérivation gauche**
  - **S → aYZ → aZ → ????**

=> impasse : Z ne se dérive pas en c, donc ac n'appartient pas à L(G)

B - Analyse syntaxique DGD

II - Analyseur DGD procédural - Points de génération

- L'analyse DGD par automate à pile peut être mise en œuvre par **procédures**
  - **Analyseur procédural**

- **Principes :**
  - **À chaque non-terminal => 1 procédure**
  - Gestion de la pile par le langage de programmation (Java par exemple) : notion de pile des appels de procédures

B - Analyse syntaxique DGD

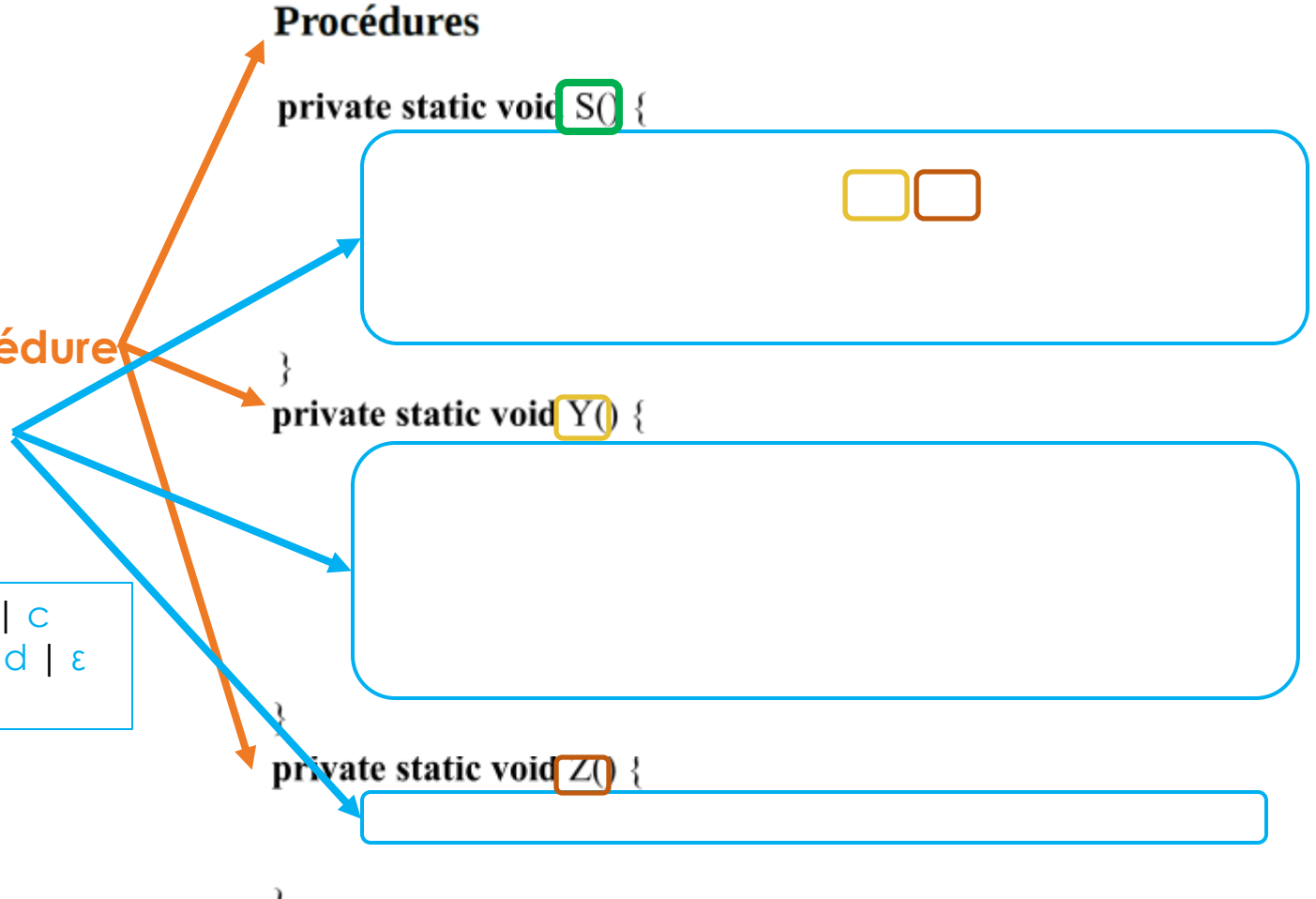
II - Analyseur DGD procédural - Points de génération

Analyseur procédural :

- **Ex : Analyseur de l'ex précédent**
  - **Dérivation à partir de l'axiome**
  - **À chaque non-terminal => 1 procédure**
  - **À chaque terminal possible en partie droite => un *case***

```
Lex.lireSymb();
S(); // appel procédure de l'axiome
if (finChaine())
   System.out.println("succès") ;
else
    System.out.println("échec") ;
```

S → a Y Z | c
Y → a Y | d | ε
Z → b



B - Analyse syntaxique DGD

II - Analyseur DGD procédural - Points de génération

Analyseur procédural :

- **Ex : Analyseur de l'ex précédent**
  - **Dérivation aab**

```
Lex.lireSymb();
S(); // appel pro
if (finChaine())
   System.out.println("succès") ;
else
   System.out.println("échec") ;
```

1- Lecture du a => Lex.itemCourant = a
2- Appel de S()

Procédures

```
private static void S() {
        switch (Lex.itemC
                case code_a : Lex.lireSymb(); Y(); Z(); break;
                case code_
                default : er
        }
}
private static void Y() {
        switch (Lex.itemC
                case code_a : Lex.lireSymb(); Y(); break;
                case code_d : Lex.lireSymb(); break;
                case code_b : break;
                default : e
        }
}
private static void Z() {
        if (Lex.itemCourant == code_b) Lex.lireSymb();
        else erreur();
}
```

3- Lecture du a => itemCourant = a
4- Appel de Y()

8- Appel de Z()

11- Sortie de S()

5- Lecture du b => itemCourant = b
6- Appel de Y()

7- Sortie de Y()

9- Lecture => itemCourant = fin_chaine
10 - Sortie de Z()

B - Analyse syntaxique DGD

II - Analyseur DGD procédural - Points de génération

Analyseur procédural :

- **RQ :** ce mode de programmation

  => Déterminisme pour les *case*

  ~ 1 item ⇔ unique *case* par procédure

=> *C'est la grammaire qui doit vérifier ces conditions*

Notion de grammaire LL(1) / LL(k) … bientôt

```
Lex.lireSymb();
S(); // appel procédure de l'axiome
if (finChaine())
   System.out.println("succès") ;
else
    System.out.println("échec") ;
```

Procédures

```
private static void S() {
        switch (Lex.itemCourant) {
                case code_a : Lex.lireSymb(); Y(); Z(); break;
                case code_c : Lex.lireSymb(); break;
                default : erreur();
        }
}
private static void Y() {
        switch (Lex.itemCourant) {
                case code_a : Lex.lireSymb(); Y(); break;
                case code_d : Lex.lireSymb(); break;
                case code_b : break;
                default : erreur();
        }
}
private static void Z() {
        if (Lex.itemCourant == code_b) Lex.lireSymb();
        else erreur();
}
```

B - Analyse syntaxique DGD

II - Analyseur DGD procédural - Points de génération

- Appels à des procédures :
  - Pour la reconnaissance de la chaîne
  - Pour réaliser des actions (calcul, affichage, …)

- Pour les actions :
  - Insertion de points de génération dans les règles de la grammaire

- Mise en œuvre pour le TP :
  - Une classe *PtGen* contenant une méthode *executer(int numPtGen)*

B - Analyse syntaxique DGD

II - Analyseur DGD procédural - Points de génération

- Pour les actions :
  - Insertion de points de génération dans les règles de la grammaire
  - **Mise à jour** de la procédure associée à *S*

- Ex. de 6 points de génération insérés :
  - S → Y **1** b
    | Z Y **2** d **3**
  - Y → a **2** Y | e **5** **6** | ε
  - Z → **4** c

Avec, par ex, code point de génération
1: System.out.println("fin analyse premier Y");

```
private static void S() {
    switch (Lex.itemCourant) {
      case code_a :
      case code_e :
      case code_b : Y() ; PtGen.executer(1); // règle S -> Yb
                    if (Lex.itemCourant == code_b) Lex.lireSymb();
                    else erreur();
                    break;
      case code_c : Z() ; Y() ; PtGen.executer(2); // règle S -> Z Y d
                    if (Lex.itemCourant == code_d) Lex.lireSymb();
                    else erreur();
                    PtGen.executer(3); break ;
      default : erreur();
    }
}
```

B - Analyse syntaxique DGD

II - Analyseur DGD procédural - Points de génération

- Exercice 2 :

  Pour cette grammaire, réécrire les procédures associées à Y et Z avec les appels aux actions.

- Ex. de 6 points de génération insérés :
  - S → Y **1** b
    | Z Y **2** d **3**
  - Y → a **2** Y | e **5** **6** | ε
  - Z → **4** c

```
public static void S() {
    switch (Lex.itemCourant) {
      case code_a :
      case code_e :
      case code_b : Y() ; PtGen.executer(1); // règle S -> Yb
                    if (Lex.itemCourant == code_b) Lex.lireSymb();
                    else erreur();
                    break;
      case code_c : Z() ; Y() ; PtGen.executer(2); // règle S -> Z Y d
                    if (Lex.itemCourant == code_d) Lex.lireSymb();
                    else erreur();
                    PtGen.executer(3); break ;
      default : erreur();
    }
}
```

B - Analyse syntaxique DGD

II - Analyseur DGD procédural - Points de génération

- Exercice 2 :

  Pour cette grammaire, réécrire les procédures associées à Y et Z avec les appels aux actions.

- Ex. de 6 points de génération insérés :
  - S → Y **1** b |

    Z Y **2** d **3**
  - Y → a **2** Y | e **5** **6** | ε
  - Z → **4** c

```
private static void Z() {
    PtGen.executer(4);
    if (Lex.itemCourant == code_c) Lex.lireSymb() ;
    else erreur();
}
```

```
private static void Y() {
    switch (Lex.itemCourant) {
      case code_a : Lex.lireSymb() ; PtGen.executer(2); Y() ; break ; // règle Y -> a Y
      case code_e : Lex.lireSymb() ; PtGen.executer(5);                // règle Y -> e
                    PtGen.executer(6); break ;
      case code_b : // règle Y -> ε       avec b ou d "derrière" Y
      case code_d : break ;
      default : erreur();
    }
}
```

B - Analyse syntaxique DGD

II - Analyseur DGD procédural - Points de génération

- Principe de fonctionnement :

  Exécution de chaque action lorsqu'elle est atteinte par l'analyse DGD

- **Arbre syntaxique décoré**

  S

- Ex :
  - S → Y **1** b | Z Y **2** d **3**
  - Y → a **2** Y | e **5** **6** | ε
  - Z → **4** c

  - Analyse de *cad* :
    - *Dérivation :*
    - *S*
    - 4 c a 2 2 d 3 = exécuter 4, consommer c, consommer a, exécuter 2, exécuter 2, consommer d, exécuter 3.

B - Analyse syntaxique DGD

II - Analyseur DGD procédural - Points de génération

- Principe : Exécution de chaque action lorsqu'elle est atteinte par l'analyse DGD

- **C'est bien l'emplacement des points de génération dans la grammaire qui décide de l'ordre d'exécution des actions**
  - Ex : ordre d'affichage préfixé (début de règle) ; post-fixé (fin de règle), …

B - Analyse syntaxique DGD

II - Analyseur DGD procédural - Points de génération

- Exercice 3 : *(= préparation au TP sous ANTLR)*
- Placer des points de génération (et écrire les traitements correspondants) dans la grammaire ci-dessous afin **d'afficher l'expression en notation polonaise post fixée** à raison d'un item par ligne.
- Grammaire sur V_T= {+, -, *, div, (,), nbentier, ident},
  Et axiome <exp4> :
  - <exp4> → <exp5> ('+' <exp5> | '-' <exp5>)*
  - <exp5> → <primaire> ('*' <primaire> | 'div' <primaire>)*
  - <primaire> → nbentier | ident | '(' <exp4> ')'
- Affichage attendu de 10 + 3 * (5 div 2) - 1

  10

  3

  5

  2

  div

  *

  +

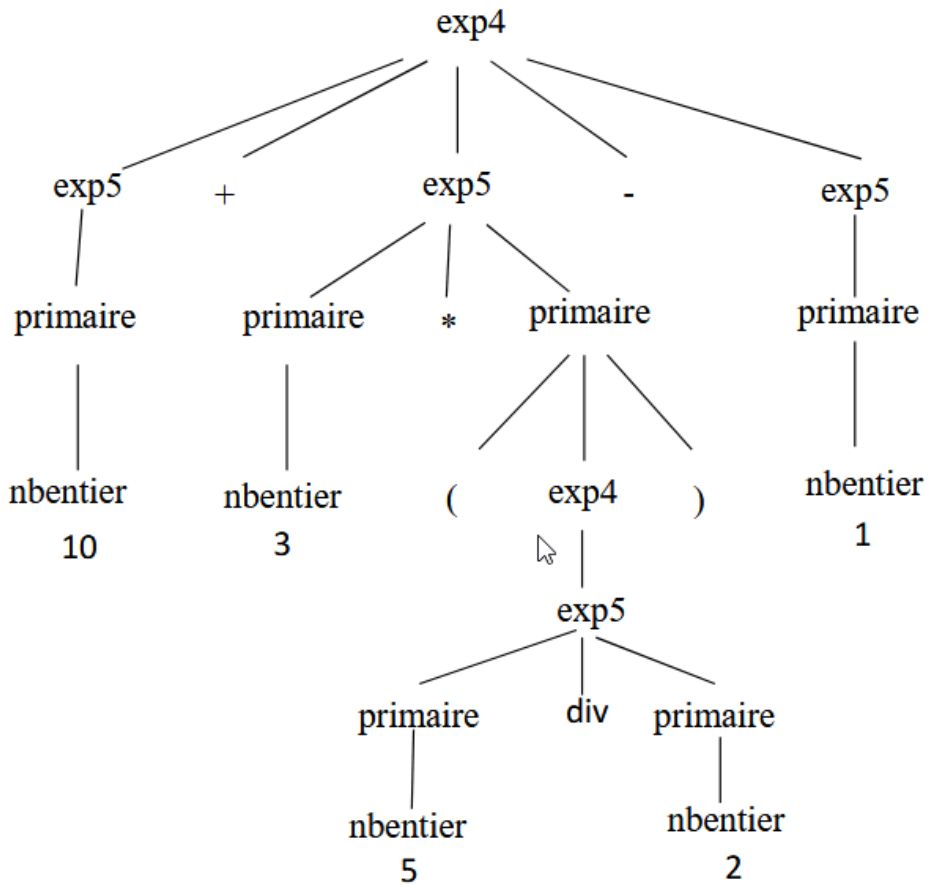
  1

  -

B - Analyse syntaxique DGD

II - Analyseur DGD procédural - Points de génération

- Exercice 3 : *(= préparation au TP5 avec ANTLR)*
- Placer des points de génération (et écrire les traitements correspondants) dans la grammaire ci-dessous afin **d'afficher l'expression en notation polonaise post fixée** à raison d'un item par ligne.
- Grammaire sur V_T= {+, -, *, div, (,), nbentier, ident},
  Et axiome <exp4> :
  - <exp4> → <exp5> ('+' <exp5> | '-' <exp5>)*
  - <exp5> → <primaire> ('*' <primaire> | 'div' <primaire>)*
  - <primaire> → nbentier | ident | '(' <exp4> ')'



arbre de dérivation