Durée : 3h. **Tous documents autorisés**. **Sujet** : 3 pages

Exercice 1 : Grilles de mots croisés

On s'intéresse dans cet exercice à la représentation textuelle de grilles de mot croisés en cours de remplissage. Une grille possède un nombre de lignes l et un nombre de colonnes c. Elle est constituée de $l * c$ cases. Une case peut être soit noire (représentée par `*`), soit blanche (représentée par `+`), soit elle contient une lettre. La grille est décrite ligne par ligne, chaque ligne étant terminée par un retour-chariot (`CR`). La description indique les dimensions de la grille (nombre de lignes l puis nombre de colonnes c, entiers strictement positifs).

On utilise la grammaire G_g suivante, d'axiome *grille*, et de terminaux V_T = { `*`, `+`, `CR`, `lettre`, `entier`} :

grille → *dim listLignes*
listLignes → *ligne* `CR` | *ligne* `CR` *listLignes*
ligne → *case* | *case ligne*
case → `*` | `lettre` | `+`
dim → `entier entier CR`

Par exemple la grille ci-dessus est représentée par le mot `entier entier CR * + CR + lettre CR`, tels que les deux symboles `entier` portent la valeur 2 et `lettre` porte la valeur `'u'`.

Q 1.1 : Donner un arbre syntaxique pour ce mot. □

1.1 : Attribution

Pour les attributions qui suivent, on indiquera pour **chaque attribut** à quel symbole il est attaché, s'il est fixé par l'analyseur lexical le cas échéant, et s'il est hérité ou synthétisé. On écrira les actions en **pseudo-java**.

On souhaite attribuer la grammaire pour vérifier que les dimensions indiquées dans la description de la grille sont cohérentes avec le nombre de cases décrites, avec **levée d'une exception** de type `DimException` en cas d'incohérence. Par exemple pour la grille ci-dessus (de taille 2 par 2), si la première ligne contenait 3 descriptions de cases et la seconde 1 description de case alors il y aurait incohérence pour le nombre de colonnes (bien que le nombre total de cases soit correct). La grille serait aussi incohérente si elle contenait 3 lignes.

Q 1.2 : Dans un premier temps on considère juste les productions :
grille → *dim listLignes*
listLignes → *ligne* `CR` | *ligne* `CR` *listLignes*
dim → `entier entier CR`
Attribuer cette portion de grammaire pour vérifier que le nombre de lignes de la description est cohérent. □

Q 1.3 : On considère maintenant la grammaire G_g complète. Attribuer G_g pour vérifier que le nombre de colonnes de la description est cohérent. □

1.2 : Analyse syntaxique et sémantique descendante

On souhaite réaliser l'analyse syntaxique des grilles par un analyseur descendant.

Q 1.4 : La grammaire G_g n'est pas LL(1) : pourquoi ? (question ne nécessitant normalement pas de construire une table d'analyse) □

Q 1.5 : Donner les productions qui manquent à la grammaire ci-dessous (production 5 à compléter, productions et non-terminaux à ajouter éventuellement) pour obtenir une grammaire G_2 équivalente à G_g, mais qui est LL(1).

1 *grille* → *dim listLignes*
2 *listLignes* → *ligne* `CR` *suiteList*
3 *suiteList* → ϵ
4 *suiteList* → *listLignes*
5 *ligne* → ...`//` **à compléter**
6 *case* → `*`
7 *case* → `lettre`
8 *case* → `+`
9 *dim* → `entier entier CR`
... ...`//` **à compléter éventuellement**

□

Q 1.6 : Calculer les ensembles *Premier* et *Suivant* de la grammaire G_2. Si vous n'avez pas su répondre à la question 1.5, vous pouvez quand même effectuer une partie des calculs. □

Q 1.7 : Construire la table d'analyse de G_2 et justifier en utilisant cette table que G_2 est LL(1). Vous pouvez utiliser les numéros de productions de la question 1.5. Si vous n'avez pas su répondre à la question 1.5, vous pouvez quand même construire une partie de la table. □

On souhaite coder un analyseur syntaxique récursif descendant pour G_2. On suppose donné un type énuméré Java `TypeSymboles` définissant comme des constantes statiques (ou comme un `enum`) les symboles terminaux de la grammaire `entier`, `lettre`, `CR`, `NOIR` (décrit `*`) et `BLANC` (décrit `+`), plus le marqueur de fin de fichier `EOF`. On s'autorisera à abbrévier `TypeSymboles` en `TS`, on écrira par exemple `TS.CR`. On utilisera la méthode `public void consommer(TypeSymboles t) throws ScannerException` et l'attribut `courant` de type `TypeSymboles` vus en cours.

Q 1.8 : Donner les méthodes (signature et corps) de l'analyseur syntaxique récursif descendant qui reconnaissent respectivement les non-terminaux *listLignes* et *suiteList* de G_2, avec levée d'une exception `ParserException` en cas d'erreur syntaxique. □

On souhaite maintenant effectuer une analyse sémantique en parallèle de l'analyse syntaxique LL(1) de G_2.

Q 1.9 : Expliquer comment adapter à G_2 l'attribution de la question 1.3 pour vérifier la cohérence du **nombre de colonnes** d'une grille. Si vous n'avez pas su répondre à la question 1.5 vous pouvez considérer comme calculés le(s) attribut(s) de *ligne* dont vous avez besoin. □

Q 1.10 : Donner à nouveau les méthodes (signature et corps) de l'analyseur sysntaxique récursif descendant qui reconnaissent respectivement les non-terminaux *listLignes* et *suiteList* de G_2, en vérifiant en même temps la cohérence du nombre de colonnes de la grille. L'analyseur lève une exception `ParserException` en cas d'erreur syntaxique et une `DimException` en cas d'incohérence du nombre de colonnes. Vous pouvez modifier sur votre copie votre réponse à la question 1.8 en utilisant une couleur différente. □

Exercice 2 : Analyse ascendante

On considère la grammaire G d'axiome P et de terminaux { `+`, `-`, `,`, `int`} :

$P \rightarrow S\ N$
$N \rightarrow E$ `,` $I \mid I$ $\qquad$ $E \rightarrow I \mid \epsilon$
$S \rightarrow$ `+` $\mid$ `-` $\mid \epsilon$ $\qquad$ $I \rightarrow$ `int`

Q 2.1 : Donner l'automate LR-AFD associé à G. On rappelle qu'à la production $X \rightarrow \epsilon$ est associé l'item $[X \rightarrow \bullet]$ (item terminal) et que l'automate LR-AFD ne contient pas de ϵ-production. □

Q 2.2 : Identifier tous les conflits dans l'automate LR(0). □

Q 2.3 : G est-elle SLR(1) ? Justifier. □

Exercice 3 : Écriture d'une grammaire

On s'intéresse dans cet exercice à l'écriture d'une grammaire pour la représentation textuelle de diagrammes de classe et paquetage UML (version très simplifiée).

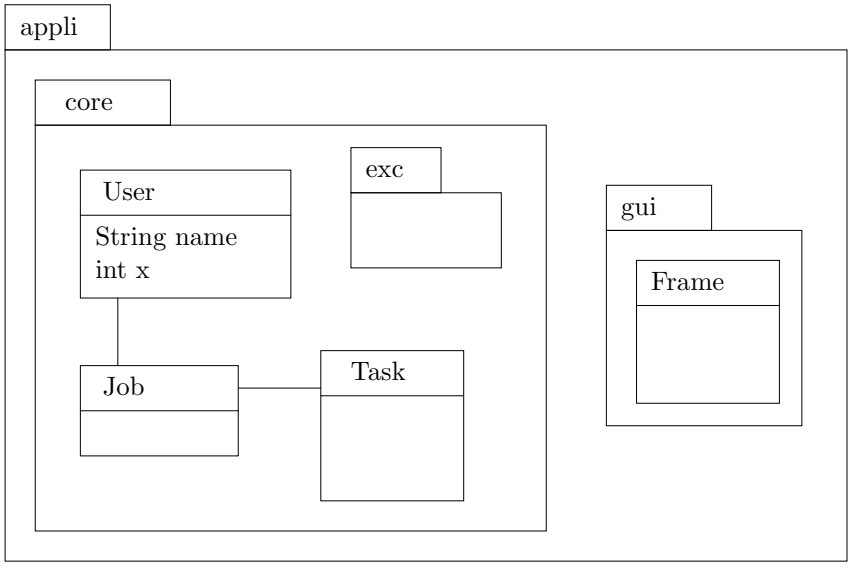
Un diagramme décrit un ensemble non vide de paquetages. Sa représentation textuelle est la liste des représentations textuelles de ces paquetages. Un paquetage est décrit par le mot-clé `package`, suivi du nom du paquetage (un identificateur Java) suivi de la description du contenu du paquetage commençant par { et terminant par }. Le contenu d'un paquetage commence par la liste éventuelle des sous-paquetages qu'il contient. Ensuite on trouve la description des classes (liste éventuellement vide de descriptions de classes) contenues dans ce paquetage, puis des relations entre ces classes (s'il y en a).

Une classe est décrite par le mot-clé `class` suivi du nom de la classe suivi entre { et } de la liste des attributs de la classe. Il n'y a pas de classes internes. Un attribut est représenté par son nom (un identificateur Java) suivi de `:` puis de son type (`String` ou `Integer`) terminé par `;`.

Le mot-clé `%relation` suivi de { permet de décrire une liste non vide de relations entre classes, terminée par }. Une relation est représentée par deux noms de classe séparés par `-`, `<->`, `->` ou `<-`, et est terminée par `;`.

À titre d'exemple, au diagramme de droite correspond la description de gauche :

```
package appli {
  package core {
    package exc {}
    class User {
      name : String;
      x : Integer;
    }
    class Job {}
    class Task {}
    %relation{
       User - Job;
       Job - Task;
    }
  }
  package gui {
    class Frame {}
  }
}
```



Q 3.1 : Donner une grammaire algébrique qui engendre l'ensemble des descriptions de tous les diagrammes décrits ci-dessus. Entourer l'axiome et souligner les terminaux. □

Q 3.2 : Le langage décrit par la grammaire est-il régulier ? Justifier intuitivement. □

Q 3.3 : Est-il possible qu'une classe ait le nom `package` ? Le nom `relation` (sans `%`) ? Justifier en expliquant le comportement de l'analyseur lexical. □