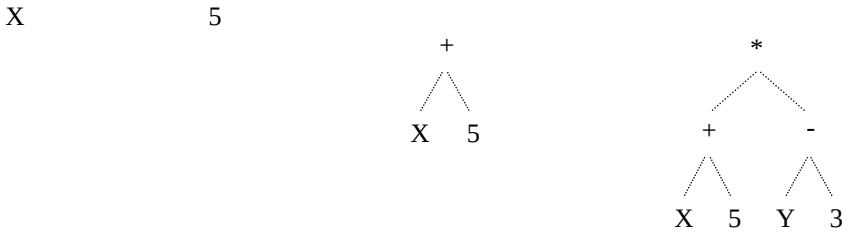
Examen

3 heures - documents écrits autorisés
vendredi 5 septembre 2003

On veut concevoir un évaluateur d'expressions symboliques. Différents traitements pourront être appliqués sur ces expressions tels que l'affichage, l'évaluation, la simplification, etc. Une expression a la forme suivante :

```
<Expression>        ::=   <Atome> | <ExpressionBinaire>
<Atome>             ::=   <Constante> | <Variable>
<Constante>         ::=   1 entier (int en Java)
<Variable>          ::=   1 chaîne (String  en Java)
<ExpressionBinaire> ::=   <Plus> | <Moins> | <Mult> | <Div>
<Plus>              ::=   <Expression> + <Expression>
<Moins>             ::=   <Expression> - <Expression>
<Mult>              ::=   <Expression> * <Expression>
<Div>               ::=   <Expression> / <Expression>
```

Voici des exemples de telles expressions représentées comme d'usage sous forme d'arbre :



Q 1 . Proposez un graphe d'héritage de classes permettant la représentation de telles expressions.

Affichage L'affichage d'une expression est réalisée en "demandant" à chacun des éléments qui constitue cette expression comment il s'affiche par appel à la classique méthode `String toString`. L'affichage des expressions binaires est encadrée par des parenthèses.

Evaluation d'une expression L'évaluation des expressions est réalisée en parcourant l'expression. Ainsi chaque élement de l'expression sait comment s'évaluer et donc toutes les expressions doivent accepter l'envoi de message suivant :

```
public Constante evaluer (Memoire mem);
```

L'évaluation numérique d'une expression doit retourner une constante représentant la valeur de cette évaluation. Le paramètre `Memoire` est utilisé par les objets `Variable`. La classe `Memoire` implémente en faitun dictionnaire stockant et centralisant la valeur associée à chaque variable.

Q 2 . Donnez le code de la classe `Memoire` qui permet de savoir pour un symbole de variable quelle est l'*expression constante* qui définit la valeur associée à ce symbole. Il faut bien sûr pouvoir modifier une mémoire et accéder aux valeurs des constantes. On supposera pour tout le sujet que les variables sont représentées par une chaîne sans espace commençant par une lettre.

Q 3 . Donnez le diagramme UML de chacune des entités (classes ou interfaces) de votre graphe d'héritage de la question 1, vous préciserez les attributs, constructeurs et méthodes (sans répéter les éventuelles méthodes héritées). Il est inutile de répéter les relations entre les différentes entités.

Q 4 . Donnez le code java des classes (et de totue les classes interfaces utilisées par celles-ci) permettant de gérer les expressions <`Constante`>, <`Variable`> et <`Mult`>.

Q 5 . Un interprète possède une mémoire et dispose de la seule méthode :

```
public void evalue(Expression exp)
```

Elle a pour effet d'afficher le message :

`l'expression` *`exp`* `a pour valeur` *`valeur`*

où *`exp`* est l'expression et *`valeur`* est la valeur de l'expression selon la mémoire de l'interprète.

Donnez le code java d'une classe `Interprete`.

Q 6. Définissez une classe `ExpressionFactory` qui appliquera le design pattern factory en fournissant une méthode telle que :

```
Expression build(String expression)
```

qui à partir d'une chaîne de caractères représentant une expression en notation infixée (la notation usuelle) construit l'objet `Expression` correspondant (on supposera que la chaîne de caractères correspond toujours effectivement à une expression).

Les expressions binaires sont parenthésées et des espaces séparent les différents éléments d'une expression.

Voici des exemples de chaînes de caractères valides :

```
"(1 + X) * 2"
"(2 + (Y * X)) - (X + 1)"
"2"
"X"
"(1 + X)"
```

Q 7. Ajoutez à la classe `Interprete` une méthode `public void evalue(String expression)` qui a le même comportement que la méthode `evalue(Expression exp)` mais à partir d'une expression donnée sous forme de chaîne supposée correcte.

Q 8. Indiquez ce qu'il faut faire (ajout, modification, etc.) pour prendre en compte les expressions unaires (+, -) et l'expression binaire puissance (opérateur ^). Vous donnerez les diagrammes de classe des nouvelles entités (pas de code demandé).

La grammaire est alors modifiée ainsi :

```
<Expression>        ::=   <Atome> | <ExpressionBinaire> | <ExpressionUnaire>
<Atome>             ::=   <Constante> | <Variable>
<Constante>         ::=   1 entier (int en Java)
<Variable>          ::=   1 chaîne (String  en Java)
<ExpressionUnaire>  ::=   <PlusUnaire> | <MoinsUnaire>
<PlusUnaire>        ::=   + <Expression>
<MoinsUnaire>       ::=   - <Expression>
<ExpressionBinaire> ::=   <Plus> | <Moins> | <Mult> | <Div> | <Puis>
<Plus>              ::=   <Expression> + <Expression>
<Moins>             ::=   <Expression> - <Expression>
<Mult>              ::=   <Expression> * <Expression>
<Div>               ::=   <Expression> / <Expression>
<Puis>              ::=   <Expression> ^ <Expression>
```

Extraits de javadoc

`java.lang.String`

`String trim()` Returns a copy of the string, with leading and trailing whitespace omitted.

`int indexOf(String str)` Returns the index within this string of the first occurrence of the specified substring.

`String substring(int beginIndex)` Returns a new string that is a substring of this string.

`String substring(int beginIndex, int endIndex)` Returns a new string that is a substring of this string.

`java.util.StringTokenizer`

`public StringTokenizer(String str, String delim)` Constructs a string tokenizer for the specified string. The characters in the delim argument are the delimiters for separating tokens. Delimiter characters themselves will not be treated as tokens.

`public StringTokenizer(String str, String delim, boolean returnDelims)` Constructs a string tokenizer for the specified string. All characters in the delim argument are the delimiters for separating tokens.

If the `returnDelims` flag is `true`, then the delimiter characters are also returned as tokens. Each delimiter is returned as a string of length one. If the flag is `false`, the delimiter characters are skipped and only serve as separators between tokens.

`int countTokens()` Calculates the number of times that this tokenizer's `nextToken` method can be called before it generates an exception.

`boolean hasMoreTokens()` Tests if there are more tokens available from this tokenizer's string.

`String nextToken()` Returns the next token from this string tokenizer.