Polytech Paris-Saclay ET4Info Compilation 2020 - 2021

TD Grammaires attribuées

Exercice 1 : On considère la grammaire suivante qui décrit des nombres flottants écrits en base 2. Les parties fractionnaires et entières sont séparées par une virgule. Le non-terminal `L` représente donc une liste de bits.

```
N ::= L
    | L , L
L ::= L B
    | B
B ::= 0
    | 1
```

1. Donnez un système d'attributs sémantiques qui permette d'évaluer la valeur en décimal d'un nombre décrit par cette grammaire. Pour chaque attribut, précisez s'il est hérité ou synthétisé, sa définition informelle ainsi que les règles de calcul que vous associez à chaque règle de la grammaire.
2. Annotez l'arbre syntaxique pour le mot *w* = `101,0111` ci-dessus avec les valeurs de chaque attribut.

La valeur attendue de l'attribut `résultat` à la racine de l'arbre syntaxique est `5,4375`
(`5,4375` = $1*2^2 + 0*2^1 + 1*2^0 + 0*2^{-1} + 1*2^{-2} + 1*2^{-3} + 1*2^{-4}$)

Exercice 2 : Soit la grammaire suivante qui a pour langage associé l'ensemble des expressions régulières sur un alphabet donné. Une lettre de l'alphabet est désignée par le token `lettre`. On note l'union par `+`, l'itération par `*` et la concaténation sans opérateur, par juxtaposition d'expressions. Pour éviter toute ambiguïté entre les deux usages possibles de ε (pour indiquer une partie droite de production vide ou pour désigner l'expression régulière vide) on note par le token `eps` l'expression régulière vide (du point de vue de la grammaire il s'agit donc d'un symbole terminal, comme `'+'` ou `lettre`)

```
S ::= S '+' T | T
T ::= T T | T '*' | F
F ::= lettre | eps | ( S )
```

Dans ce langage l'expression « un `a` optionnel », où `a` est une instance de `lettre`, s'écrirait donc `a + eps`.

Pour une expression régulière de la grammaire ci-dessus, on veut construire un automate fini non-déterministe qui reconnaît le langage associé à l'expression. On impose de coder l'ensemble `Q` des états de l'automate par un **intervalle** d'entiers `[0..N]`, **`0` étant forcément l'état initial et `N` l'unique état de satisfaction**. Les transitions sont exprimées par une liste `L` de triplets `(i,a,j)` avec `i,j` dans `Q` et `a` une instance de `lettre` ou bien ε pour une transition à vide.

*Exemple : **un** automate possible pour l'expression* `a + b` *est :*

`N = 5` et
`L` = `{(0, ε, 1), (0, ε 3), (2, ε, 5), (4, ε, 5),`
`(1, a, 2), (3, b, 4) }`.

1. Donnez pour la grammaire augmentée de `A ::= S` un système d'attributs pour calculer un automate associé à une expression régulière. Votre système devra permettre de disposer des attributs suivants à la racine `A` de l'arbre syntaxique de l'expression:
   - le numéro du dernier état produit (`N` dans la notation ci-dessus)
   - la liste des transitions (`L` dans la notation ci-dessus)

Pour chaque attribut, indiquerez s'il est synthétisé ou hérité et donnez ses règles de calcul. On fera attention à ce qu'un même numéro d'état ne soit pas utilisé de façon erronée dans des sous-automates distincts.

2. Donnez le résultat de l'évaluation de l'évaluation de vos attributs pour le mot `a a * + eps`

Exercice 3 : soit une instruction « par cas » dont la structure est décrite informellement par :

```
case E0 of
     when E1 .. E2   => I1
     when E3 .. E4   => I2
…    …    …
     when E2n-1 .. E2n => In          n ≥ 0
     others              => In+1      Cette clause est optionnelle ; si elle
end case                              présente, elle est forcément à la fin
```

Les `Ei`, `0≤i≤2n`, sont des expressions à valeurs entières et les I_i sont des instructions arbitraires. L'expression E_0 est l'expression de contrôle et les paires d'expressions des clauses **`when`** définissent les bornes d'intervalles de valeurs. L'exécution de cette construction consiste à évaluer l'expression E_0 et, suivant la valeur obtenue, à exécuter l'instruction associée à l'intervalle auquel la valeur appartient. Si aucun intervalle ne convient, si la clause **`others`** est présente l'instruction associée est exécutée, sinon l'instruction par cas est sans effet. Tous les intervalles de valeurs doivent être disjoints.

1. Donnez une grammaire pour cette construction. On notera, sans les détailler, par `E` une expression entière et par `I` une instruction(dont l'instruction par cas). On prendra garde au fait que les instructions par cas peuvent être emboîtées.

2. Indiquez quelles sont les vérifications contextuelles à mettre en œuvre pour une telle construction. Les bornes des intervalles doivent-elles être statiques ou peuvent-elles être dynamiques ? Justifiez votre réponse.

Le langage Ada demande que dans une telle instruction, toute valeur soit couverte une fois et une seule par un des intervalles ou par la clause `others`. Donnez un système d'attributs pour vérifier si une instruction par cas est bien fondée au sens ci-dessus. *Pour simplifier vos expressions de calcul d'attributs vous pouvez faire appel à des fonctions auxiliaires, en donnant pour chacune d'entre elles son profil et ce qu'elle est supposée faire, sans la programmer explicitement.*

Exercice 4 : On considère une version très simplifiée des exceptions dans un langage à la Java. La syntaxe complète d'une déclaration de méthode suit le format ci-dessous :

typeRetour nom `(`*paramètres*`)` `throws` *exceptions* `{` *instructions* `}`

Une méthode a une liste éventuellement vide de paramètres et son corps est une séquence éventuellement vide d'instructions. La clause optionnelle `throws` *exceptions* doit déclarer **toutes** les exceptions que la fonction est susceptible de lever dynamiquement (il n'y a pas de cas particulier comme avec les hiérarchies `RunTimeException` et `Error` en Java).

Parmi les instructions du langage, on ne considère que les instructions suivantes

- on abstrait en `A` toutes les instructions dont on suppose qu'elles ne lèvent pas d'exceptions.
- `throw new C;` où `C` représente un nom de classe
- l'instruction `if then else` classique, dans laquelle les instructions des parties `then` et `else` sont obligatoires et peuvent ou non lever des exceptions.

- l'appel à une méthode: `e.nom-methode(arguments);` où `e` est une expression qui représente le destinataire du message.
- l'instruction de traitement d'exceptions

  `try {` *liste éventuellement vide d'instructions* `}`
  `catch (`classe*Id varId*`) {` *liste éventuellement vide d'instructions* `}`
  …
  `catch (`*classeId varId*`) {` *liste éventuellement vide d'instructions* `}`

Il y a au moins un bloc `catch` après un `try`. Les instructions dans les `catch` peuvent contenir des instructions arbitraires dont des `throw` ou d'autres blocs `try-catch` . Une exception levée dans un bloc `catch` n'est **pas** rattrapée dans un `catch` associé au même `try`, quel que soit l'ordre dans lequel les clauses `catch` apparaissent. On ne décrira **pas** la sous-grammaire des expressions et dans un premier temps **on considère que les expressions ne lèvent pas d'exceptions.**

1. Donnez une **grammaire non-ambiguë** pour ces déclarations de méthodes.

Ci-dessous, pour chaque attribut utilisé, donnez son nom, sa nature héritée ou synthétisée et ses règles de calcul. Précisez informellement comment l'environnement est transmis entre les règles. Les règles peuvent utiliser des fonctions auxiliaires dont vous préciserez l'en-tête et la sémantique. On suppose qu'on dispose d'une fonction `sousClasse(C1, C2)` qui renvoie vrai si et seulement si `C1` est le nom d'une sous-classe (au sens large) de `C2` et d'une fonction `enTete(env, g, C, C1, ...Cn)` qui cherche dans l'environnement `env` la méthode de signature `g` (`C1, ...Cn`) applicable aux objets de classe `C` et renvoie la liste des exceptions qui peuvent être levées par cette méthode. On supposera qu'une telle méthode existe bien (on suppose que ceci aura été vérifié au préalable). On dispose aussi au niveau de chaque déclaration de méthode d'un attribut `maclasse` qui donne la classe de rattachement de la méthode considérée.

2. **Donnez un système d'attribut**s qui vérifie le bon ordre des `catch` : une clause `catch(E x)` doit être située **avant** une clause `catch` pour une classe d'exception plus générale que `E`.

3. On veut contrôler que les exceptions levables par le corps d'une méthode apparaissent toutes dans sa clause `throws`, compte-tenu des relations d'héritage et des règles décrites pour les blocs `try-catch`. **Donnez le système d'attributs** correspondant, dont un attribut `bien-formé` qui vaut `vrai` si et seulement si le corps de la méthode est correct vis-à-vis de sa clause `throws`. On suppose qu'on dispose d'un attribut synthétisé `type` qui contient le type d'une expression.

4. On suppose maintenant que les expressions peuvent aussi lever des exceptions. En particulier, le corps d'une méthode `f` peut contenir un appel de méthode de la forme `e.g(e1, …, en)`, où `e` et les `ei` peuvent lever des exceptions en plus de celles du corps de `g`. Donnez les définitions d'attributs pour traiter ce nouveau cas (on ne décrira pas les autres types d'expressions).