3 Grammaires et Attributs

Les grammaires attribuées (à la Yacc) servent à construire de nouvelles structures, découvrir des caractéristiques des expressions analysées, calculer des valeurs,... durant l'analyse syntaxique. Dans un compilateur, elles servent à construire l'arbre syntaxique abstrait (AST)

EXERCICE 9

[Évaluation d'expressions préfixées]
Source : `www.irisa.fr/lande/ridoux/ENS/COMP/TD/sujet_td3et4.pdf`.

1. On considère la grammaire suivante qui décrit les expressions arithmétiques préfixées binaires :

```
exp -> chiffre | (+ exp exp) | (* exp exp)
chiffre -> 0 | 1 | ... | 9
```

   Attribuer cette grammaire de façon à calculer la valeur de l'expression analysée. Vérifier sur un exemple.

2. On généralise cette grammaire pour décrire maintenant les expressions n-aires préfixées :

```
exp -> chiffre | (+ exp sexp) | (* exp sexp)
sexp -> exp | sexp
chiffre -> 0 | 1 | ... | 9
```

   – Construire l'arbre de dérivation de + (∗ 2 3) 4 3.
   – Attribuer cette nouvelle grammaire de façon à là encore calculer la valeur de l'expression analysée.

CORRECTION :

1. Expressions binaires.
   – Chacun des non-terminaux va avoir un S-attribut de type entier, qui sera sa valeur entière. Les attributs étant tous calculés des fils vers le père, ce sera une grammaire S-attribuée.

Non terminal	Attribut	type
chiffre	chiffre↑val	entier
exp	exp↑val	entier

   – Il reste à écrire les règles de calcul :

Règle	Actions
`chiffre->0`	{chiffre↑val← 0}
`exp->chiffre`	{exp↑val ←chiffre↑val }
`exp0->(*exp3 exp4)`	{exp0↑val ← exp3↑val * exp4↑val}

   Les autres régles sont similaires.
   – Vérification rapide sur (+2(∗38)).

2. Expressions n-aires.
   – Là encore, les attributs vont être des entiers :

Non terminal	Attribut	type
chiffre	chiffre↑val	entier
exp	exp↑val	entier

– On commence à écrire les règles :

Règle	Actions
`chiffre->0`	{chiffre↑val← 0}
`exp->chiffre`	{exp↑val ←chiffre↑val }
`exp0->(*exp3 sexp)`	{exp0↑val ← exp3↑val * sexp↑val}
`sexp->exp`	{sexp↑val ← sexp↑val}

Jusque là, tout va bien, sauf que lorsque l'on regarde la règle `sexp->exp sexp`, on ne sait pas si il faut additionner ou multiplier. On va donc ajouter un H-attribut (hérité) pour donner l'information du père vers les fils que le `sexp` que l'on est en train de dériver provient d'un + ou un * : l'attribut sexp↓op qui sera un entier (1 si +, 2 si *) par exemple. Il reste à modifier les règles pour prendre en compte ce nouvel attribut hérité :

Règle	Actions
`chiffre->0`	{chiffre↑val← 0}
`exp->chiffre`	{exp↑val ←chiffre↑val }
`exp0->(*exp3 sexp)`	{exp0↑val ← exp3↑val * sexp↑val ; **sexp↓op← 2**}
`sexp->exp`	{sexp↑val ← sexp↑val}
`sexp0->exp sexp2`	{sexp↑val ← exp↑val $\bowtie$ sexp2↑val, $\bowtie = \begin{cases} + & \text{si } sexp0 \downarrow op == 1 \\ * & \text{sinon} \end{cases}$, sexp2↓op←sexp0↓op}

et ensuite, on laisse la magie opérer !

Exercice 10

[Évaluation des nombres flottants]
On considère la grammaire pour l'écriture des nombres flottants :

```
S -> X E D
X -> + | - | eps
E -> BE | B
B -> 0 | 1 | ... | 9
D -> .E
```

– S-attribuer la grammaire pour calculer la valeur d'un mot de $L(G)$.
– Même question mais avec un ou plusieurs attribus hérités.

Correction : En attente du source d'erwan

Exercice 11

[Réécriture d'une expression sous forme préfixée]
On considère la grammaire des expressions arithmétiques (simples) suivante :

```
E  -> E + F | F
F  -> F * T | T
T  -> (E) | id
```

Attribuer la grammaire pour réécrire l'expression sous forme préfixée. La chaîne "2+5*8" se réécrira donc en la chaîne "+ 2 5 8". Vérifier que les attributs font bien ce que l'on veut dans le cas de la chaîne (2 + 5) ∗ 8.

Correction : Cette fois les attributs ne sont plus des entiers, mais vont être des chaînes de caractères, que l'on va concaténer à l'aide de l'opérateur ^ :

Non terminal	Attribut	type
E	E↑val	string
F	F↑val	string
T	T↑val	string

Et donc, les règles :

Règle	Actions
`T->id`	{T↑val← "id"}
`T->E`	{T↑val← ")"^E↑val^")"}
`F->T`	{T↑val← ")"^E↑val^")"}
`F->F*T`	{F↑val← *F↑val^" "^T↑val}

Exercice 12

[Construction de l'arbre abstrait]
Soit L le langage décrit par la grammaire suivante :

```
E  -> E + F | F
F  -> F * T | T
T  -> (E) | id
```

Attribuer la grammaire pour construire l'arbre de l'expression. On suppose que l'on dispose des fonctions suivantes :

- `creerNoeud : id -> Arbre` qui construit l'arbre `Noeud(id)`.
- `creerArbre : val,fd,fg -> Arbre` qui construit l'arbre de racine val et fd,fg les deux sous-arbres.

Correction : En attente du source d'Erwan

Exercice 13

[Recherche de profondeur de définition]
Source : `www.irisa.fr/lande/ridoux/ENS/COMP/TD/sujet_td3et4.pdf`
On considère les programmes décrits par la grammaire suivante :

```
programme   ->   bloc
bloc        ->   {dec ; sint}
dec         ->   TKVAR suite-ident
suite-ident ->   ident,suite-ident | ident
sinst       ->   opcode (suite-arg) | bloc
suite-arg   ->   suite-arg,ident | ident
```

où ident et opcode désignent respectivement un identificateur quelconque non réservé et un identificateur de fonction prédéfinie, et TKVAR le mot clef « var ».

- Donner quelques exemples de programmes.
- On souhaite associer à chaque occurrence d'identificateur sa profondeur (≥ 1) et son rang (≥ 1) de déclaration. Ainsi, sur l'exemple suivant :

```
{ var a,b,c ; // a(1,1), b(1,2), c(1,3)
  f1(a,c) ;   // a(1,1), c(1,3)
  { var d,b ;
    f2(d,c,b);
  }
  {
    var e,b,a; // e(2,1)
  ...
  }
...
}
```

  on fait figurer le couple (profondeur, rang) correspondant à chacune des occurences.
- Donner un système d'attributs permettant d'associer à chaque occurrence de *déclaration* de variable, la profondeur du bloc où elle est déclarée et le rang de cette variable dans sa liste de déclaration.
- Donner un système d'attributs permettant d'associer à chaque occurrence de *d'utilisation* d'un identificateur la profondeur et le rang de déclaration de cet identificateur.

Correction : TODO

Exercice 14

[Vérification de contraintes]
En Lisp, on définit les arbres à l'aide de la grammaire suivante :

$$arbre \rightarrow () \mid int\ arbre\ arbre$$

1. À quel arbre correspond « naturellement » l'expression 4 (3 () ()) (7 () ()) ?
2. Attribuer la grammaire pour vérifier qu'un tel arbre est un arbre binaire de recherche. (Chaque nœud de chaque sous-arbre droit est plus petit que la racine, qui est elle même plus petite que chaque feuille du sous-arbre droit).

Correction : On invente un attribut booléen `isABR` qui sera vrai ssi l'arbre en question est un ABR. Il va être un S-Attribut parce que l'on utilise les infos des sous arbres pour remonter l'information. Il faut aussi rajouter un attribut pour remonter l'information "maximum" et "minimum" de l'arbre, afin de pouvoir tester (remarque, un seul attribut qui serait la racine de l'arbre ne fonctionne pas, pourquoi ?). On prend donc les trois attributs S suivants :

Non terminal	Attribut	type
arbre	arbre↑isABR	booléen
arbre	arbre↑max	entier
arbre	arbre↑min	entier

Voici les règles associées

`arbre -> ()`	arbre↑isABR ← true arbre↑max ← $-\infty$. arbre↑min ← $+\infty$.
`arbre0 -> int arbre2 arbre3`	arbre0↑isABR ← arbre2↑isABR **and** arbre3↑isABR **and** int<arbre2↑max **and** int>arbre3↑max

Il resterait à faire en sorte de surcharger les comparaisons afin qu'elles prennent en compte $+/-\infty$.