Cours – Grammaires attribuées

Les grammaires attribuées permettent d'effectuer un calcul pendant la reconnaissance.

1 Grammaires à attributs dits synthétisés (= résultat d'un caclul)

Les non-terminaux sont encore implantés sous forme de fonctions récursives qui retournent un booléen indiquant si la chaîne de caractères passées en paramètre est reconnue par la grammaire.
Dans le cas d'une grammaire attribuée, on étend la fonction associée à chaque non-terminal pour qu'en plus du booléen elle retourne le résultat d'un calcul.

Notation La règle de production $S \to S_1.S_2$ étendu avec ses attributs r, r_1, r_2 se note :

$$S\ \{r\}\ \to S_1\ \{r_1\}\ .\ S_2\ \{r_2\}\ \ ;\ \ \{r := f(r_1, r_2)\}$$

et se lit de la manière suivante :
- Comme précédemment, le non-terminal S reconnaît une chaîne si le début est reconnu par S_1 et la suite reconnue par S_2.
- La reconnaissance par S produit un résultat r, noté entre accolades après S. De même, la reconnaissance par S_1 produit un résultat r_1 et la reconnaissance par S_2 produit un résultat r_2.
- Le résultat rendu par le non-terminal S est calculé à partir des résultats (r_1 et r_2) par l'instruction $r := f(r_1, r_2)$.

Implantation directe par des fonctions récursives

```
S : string → (string, bool, résultat)

S(ch) = (ch1, b1, r1) := S1(ch) ;
        (ch2, b2, r2) := S2(ch1) ;
        return ( b1 && b2 && ch2="" , f(r1, r2) )
```

Implantation plus élégante par fonctions utilisant les flots du langage ocaml Les flots de caractères d'ocaml (`char stream`) permettent d'éviter d'expliciter le booléen (reconnu/pas reconnu) et le passage du reste de la chaîne à analyser (cf. cours semestre 6).

OCAML

```
S : char stream → résultat

S = parse [< r1 = S1 ; r2 = S2 >] -> f(r1, r2)
```

1.1 Application : ajout d'un compteur

On ajoute un attribut $n \in \mathbb{N}$ à la grammaire pour qu'en plus de la reconnaissance on calcule le n du mot $a^n b^n$ reconnu.

$$G_1^n = \begin{cases} S\ \{n\}\ \to \epsilon\ ;\ \{n := 0\} \\ S\ \{n\}\ \to a \cdot S\ \{n'\}\ \cdot b\ ;\ \{n := n' + 1\} \end{cases}$$

Implantation sous forme d'une fonction résursive *S : string → (bool,int)* On modifie la fonction précédente S pour qu'elle retourne le nombre n tel que $S(a^n b^n) = n$.

PSEUDO CODE

```
        S ("") = (true,0)
S ("a".ch."b") = (r,n') := S(ch) ; (r,n'+1)
         S(_) =  (false,_)
```

PSEUDO CODE

```
S(ch) =
  if ch = "" then (true,0)
  else if ch = "a".ch'."b" then (r,n') := S(ch) ; (r,n'+1)
  else (false,_)
```

1.2 Application : construction de l'arbre de dérivation

Un arbre de dérivation Δ est une donnée de la forme $\mathrm{D}(\mathrm{NT}, [\Delta_1; \ldots; \Delta_n]))$ qui correspond à l'étape de dérivation

$$\begin{array}{c} \mathrm{NT} \\ \downarrow \\ \Delta_1 \ldots\ldots \Delta_n \end{array}$$

où NT est un non-terminal et $\Delta_1, \ldots, \Delta_n$ sont eux-mêmes des arbres qui représentent la suite des dérivations

L'arbre de dérivations associé à la règle $S \to S_1.S_2$ est $\begin{array}{c} S \\ \downarrow \\ \Delta_1.\Delta_2 \end{array}$ en notation graphique, c'est-à-dire $\mathrm{D}(\mathrm{S}, [\Delta_1; \Delta_2])$ pour l'implantation, où Δ_1, Δ_2 sont les arbres de dérivation produit par S_1 et S_2.

Grammaire étendue avec les attributs de construction de l'arbre de dérivation

$$G_1^{\Delta} = \begin{cases} S\ \{\Delta\}\ \to \epsilon\ ;\ \{\Delta := \begin{array}{c} \mathrm{S} \\ \downarrow \\ \epsilon \end{array}\ \} \\ \\ S\ \{\Delta\}\ \to a \cdot S\ \{\Delta'\}\ \cdot b\ ;\ \{\Delta := \begin{array}{c} \mathrm{S} \\ \downarrow \\ a.\Delta'.b \end{array}\ \} \end{cases}$$

Implantation sous forme d'une fonction résursive *S : string → (bool, Δ)*

PSEUDO CODE

```
        S ("") = (true, D(s,[]) )
S ("a".ch."b") = (r,Δ) := S(ch) ; (r, D(s,["a";Δ;"b"]))
         S (_) = (false,_)
```

PSEUDO CODE

```
S(ch) =
  if ch = "" then (true, D(s,[]))
  else if ch = "a".ch'."b"
       then (r,Δ') := S(ch) ; return (r, D(s,["a";Δ';"b"])
       else (false,_)
```

Reconnaissance et construction de l'arbre de dérivation de `"aabb"`

résultat de l'exécution de la fonction résursive `S`

```
S("aabb") = (true, D(S,["a"; D(S,["a";D(S,[""]);"b"] ;"b"])
```

l'arbre de dérivation sous forme graphique

$$
\begin{array}{lccccc}
\Delta = & & S & & & \\
 & & \downarrow & & & \\
 & \overbrace{\text{"}a\text{".} & S & .\text{"}b\text{"}}^{} & & \\
 & & \downarrow & & & \\
 & \text{"}a\text{".} & \overbrace{\text{"}a\text{".}\, S\, .\text{"}b\text{"}}^{} & .\text{"}b\text{"} & & \\
 & & \downarrow & & & \\
\text{"}a\text{"}.\text{"}a\text{".} & & \text{""} & .\text{"}b\text{"}.\text{"}b\text{"} & \equiv & \text{"}aabb\text{"}
\end{array}
$$

2 Grammaires à attributs hérités (un paramètre) et synthétisés (le résultat)

Les non-terminaux sont encore implantés sous forme de fonctions récursives qui retournent un booléen indiquant si la chaîne de caractères passées en paramètre est reconnue par la grammaire.

Dans le cas d'une grammaire attribuée, on étend la fonction associée à chaque non-terminal pour qu'en plus du booléen elle retourne le résultat d'un calcul.

Notation La règle de production $S \rightarrow S_1.S_2$ étendu avec ses attributs hérités et synthétisés est de la forme :

$$\{r\}\ S\ \{r'\}\ \rightarrow\ \{r'_1 := h(r)\}\ ;\ \{r_1\}\ S_1\ \{r'_1\}\ ;\ \{r_2 := g(r, r'_1)\}\ ;\ \{r_2\}\ S_2\ \{r'_2\}\ ;\ \{r' := f(r, r'_1, r'_2)\}$$

et se lit de la manière suivante :

- Comme précédemment, le non-terminal S reconnaît une chaîne si le début est reconnu par S_1 et la suite reconnue par S_2.
- Les attributs d'entrée dits hérités sont notés entre accolades à gauche du non-terminal ; ce sont des paramètres du non-terminal. Les attributs de sortie dits synthétisés sont notés entre accolades à droite du non-terminal ; ce sont des résultat du calcul effectué par le non-terminal.
- La reconnaissance par $S\ \{r'\}$ produit utilise l'argument r pour produire un résultat r'. De même, la reconnaissance par $S_1\ \{r'_1\}$ produit un résultat r'_1 à partir du paramètre $r_1 := h(r)$ et la reconnaissance par $S_2\ \{r'_2\}$ produit un résultat r'_2 à partir du paramètre $r_2 := g(r, r'_1)$ obtenu à partir des résultats précédents r et r'_1.
- Le résultat r' rendu par le non-terminal S est calculé à partir des résultats (r, r'_1 et r'_2) par l'instruction $r' := f(r, r'_1, r'_2)$.

Implantation directe par des fonctions récursives

PSEUDO CODE

```
S : (string,résultat) → (string, bool, résultat)

S(ch,r) = r1 := h(r) ;
          (ch1,b1,r'1) := S1(ch,r1) ;
          r2 := g(r,r'1) ;
```

```
(ch2,b2,r'2) := S2(ch1,r2) ;
r' := f(r,r'1,r'2) ;
return ( b1 && b2 && ch2="" , r' )
```

Implantation plus élégante par fonctions utilisant les flots du langage ocaml Les flots de caractères d'ocaml (`char stream`) permettent d'éviter d'expliciter le booléen (reconnu/pas reconnu) et le passage du reste de la chaîne à analyser (cf. cours semestre 6).

OCAML

```
S : char stream → résultat

S = parse [< r'1 = S1(h(r)) ; r'2 = S2(g(r,r'1)) >] -> f(r,r'1,r'2)
```

2.1 Application : calcul de l'entier correspondant à son écriture en base 10

Lecture d'une suite de caractères avec reconnaissance des nombres entiers et calcul de l'entier correspondant.
On souhaite écrire une fonction G de reconnaissance des écritures en base 10 qui retourne l'entier correspondant.

TYPE ET EXEMPLE

```
G: string → int

G ("000123400") = 123400
```

Grammaires sans attribut

$$G = \begin{cases} S \to C \cdot S' \\ S' \to \epsilon \mid C \\ C \to \texttt{"0"} \mid \texttt{"1"} \mid \ldots \mid \texttt{"9"} \end{cases}$$

Grammaires avec attributs hérités et synthétisés

$$G_s^h = \begin{cases} \{i\}\ S\ \{o\} & \to & C\ \{c\}\ .\ \{10 \times i + c\}\ S'\ \{n\}\ ;\ \{o := n\} \\ \{i\}\ S'\ \{o\} & \to & \epsilon\ ;\ \{o := i\} \\ & \mid & C\ \{c\}\ ;\ \{o := 10 \times i + c\} \\ C\ \{c\} & \to & \texttt{"0"}\ ;\ \{c := 0\} \\ & \mid & \texttt{"1"}\ ;\ \{c := 0\} \\ & \mid & \ldots\ ;\ \{c := 0\} \\ & \mid & \texttt{"9"}\ ;\ \{c := 0\} \end{cases}$$

Implantation

Type et Exemple

```
S : (string,int) → (string, bool, int)
C : string → (string, bool, int)

S(ch,i) = (ch1,b1,c) := C(ch) ;
          (ch2,b2,n) := S'(ch1, 10 * i + c) ;
          o := n ;
          return (ch2, b1 && b2, o)

S'(ch,i) = if ch = "" then return ("",true,i) ;
           else { (ch1,b1,c) := C(ch) ;
                  o := 10 * i + c ;
                  return (ch1,b1,o) ; }

C(ch) =      if ch = "0".ch1 then return (ch1,true,0)
         else if ch = "1".ch1 then return (ch1,true,1)
         ...
         else return(ch,false,_)
```

Traduction directe en Ocaml à l'aide de parser de flot Voir le cours de programmation fonctionnelle en Ocaml au semestre suivant et son utilisation dans le projet de fin d'année.

OCAML

```
S, S' : int → char stream → int
C : char stream → int

let S i =
   parse [< c = C ; n = S'(10*i+c) >] -> n

and S' i =
   parse [< >] -> i
       | [< c = C >] -> 10 * i + c

and C =
   parse [< '0' >] -> 0
       | [< '1' >] -> 1
       | ...
       | [< '9' >] -> 9
```

3 Exercices sur les grammaires attribuées

L'exercice de l'examen sur les grammaires attribuées sera une variante des exercices suivants

3.1 Grammaire des textes

Un texte est une suite de pharses. Une phrase est une suite de mots terminée par un point. Les mots sont toujours précédés d'au moins un espaces. Un mot est composé d'au moins une lettre.

Q1. Donnez la grammaire qui reconnaît les textes

Q2. Ajoutez des attributs synthétisés pour compter le nombre de mot et le nombre de lettres du texte.

3.2 Grammaire des expressions arithmétiques

On considère les expressions arithmétiques formées de constantes (dans $\mathbb{Z}$) et des opérateurs binaires $+, \times$.

Q3. Donnez la grammaire G qui décrit les expressions arithmétiques bien formées :

$3 + 2 + 5 \times 10 \in \mathscr{L}(G)$
$+ + \times\ 3\ 4 \notin \mathscr{L}(G)$

Utilisez les non-terminaux N pour les nombres, C pour les chiffres, E pour les expressions.

Q4. Montrez que la grammaire est ambigue en montrant que l'expression $3+4+5$ admet deux arbres de dérivations

Q5. Grammaire $LL1$ Donnez une version non-ambigue de la grammaire en utilisant des règles telles que $OpE \rightarrow$ `'+'`$.E$ qui permettent de décider quelle règle utiliser en se basant sur le prochain caractères.

Q6. Calculatrice en ligne Ajoutez des attributs afin que la grammaire évalue l'expression arithmétique en même temps qu'elle la reconnaît.

Q7. Hors TD Ajoutez à la grammaire précédente l'opérateur unaire $-$ qui inverse le signe d'une expression.

3.3 Grammaire des déclaration de type

Q8. Donnez une grammaire qui définit les déclarations de types du langage C de manière à admettre les déclarations :

```
int x=1;
float y,z;
int t;
float u,v=0;
```

et à rejeter les déclarations :

```
int x, int y;
int x=0,y=1;
```

Q9. Ajoutez un attibut synthétisé de manière à rendre sous la forme d'une chaîne de caractères des déclarations individualisés.

```
int x=1; float y; float z; int t; int u; float v=0;
```

Vous utiliserez l'opérateur $\_\cdot\_$ pour concaténer deux chaînes de caractères.

Q10. Modifiez la gestion des attibuts afin de rendre sous la forme d'une chaîne de caractères un programme avec des déclarations sous la forme :

```
x,t : int ;
y,z,u,v : float ;
x:=1 ;
v:=0 ;
```

c'est-à-dire avec toutes les déclarations de type au début, regroupée par type et toutes les initialisations à la fin.

3.4 Grammaire des textes html

Q11. Donnez une grammaire qui définit les textes html formés des balises suivantes

```
<html> ... <ol>  <li>...</li> </ol> ... </html>
```

où
– les pointillés sont des textes (on utilisera le non-terminal T pour faire référence à un texte) de la grammaire de l'exercice 1)
– `<ol>` indique le début d'une liste et `<li>` est un élément de la liste
La grammaire doit autorisées les listes imbriquées :

```
<ol>
  <li> ...
    <ol>
      <li>...</li>
      <li>...</li>
    </ol>
 </li>
</ol>
```

Q12. Ajoutez des attributs afin de compter

1. le nombre d'éléments de la plus longue liste
2. la profondeur maximale d'imbrication : une liste de liste de liste est de profondeur 3

L'exemple précédent retournera le résultat $(2, 2)$