ESIAL 1 — Module Mathématiques appliquées discrètes

Analyse syntaxique descendante : analyse LL(1), analyse par descente récursive

Introduction

Dans le TD précédent nous avons montré comment construire un analyseur prédictif non récursif pour une grammaire LL(1). L'objectif de ce TD est de définir ce qu'on appelle "l'analyse par descente récursive", étant donnée une grammaire LL(1), un analyseur basé sur la descente récursive consiste à définir des procédures récursives permettant de déterminer si un mot donné appartient ou non au langage engendré par la grammaire et si c'est le cas à déterminer son arbre syntaxique ou son arbre abstrait. Support de cours "Analyse syntaxique descendante" : diapositives 13 à 16.

Les notions introduites dans ce TD pourront être utilisées dans le projet du module "Structures de Données (1A)" où il est demandé d'écrire un analyseur syntaxique.

Exercice 1 *Analyseur basé sur la descente récursive*

Soit la grammaire $\mathcal{G}$ suivante définissant des expressions arithmétiques

$\mathcal{G} = (\{E, F, G\}, \{x, y, , z, +, -, (,)\}, \rightarrow, E)$

$E \rightarrow E + F \mid E - F \mid F$

$F \rightarrow -F \mid G$

$G \rightarrow x \mid y \mid z \mid (E)$

Question 1

Soient les mots $m_1 = z$, $m_2 = -x$, $m_3 = x - z$, $m_4 = -x - -z$, $m_5 = x - y - z$ et $m_6 = x - (y - z)$, montrer que ces mots sont engendrés par $\mathcal{G}$, en dessinant les arbres *syntaxiques* de ces mots.

Représentation des arbres abstraits. On considère le type *arbreAbstrait* muni des opérations suivantes :

type <u>*arbreAbstrait*</u>

$VarProp$:	$string \rightarrow arbreAbstrait$
$MoinsUnaire$:	$arbreAbstrait \rightarrow arbreAbstrait$
$Moins$:	$arbreAbstrait \times arbreAbstrait \rightarrow arbreAbstrait$
$Plus$:	$arbreAbstrait \times arbreAbstrait \rightarrow arbreAbstrait$

Question 2

Dessiner les arbres *abstraits* des mots m_i $(1 \leq i \leq 6)$ de la question précédente et donner une représentation de ces arbres à l'aide des opérations du type *arbreAbstrait*.

Question 3

Qu'est-ce qui empêche de faire une analyse LL(1) de $\mathcal{G}$? Transformer $\mathcal{G}$ et montrer que la grammaire $\mathcal{G}'$ obtenue est LL(1) (on dénotera H le nouveau non-terminal introduit).

Question 4

Mise en place de l'analyseur. Soit $\mathcal{G} = (N,\ T,\ \rightarrow,\ S)$ une grammaire LL(1). On se donne les fonctions :

- Le programme principal est donné en Algorithme 1. Il prend un mot m à analyser. Durant son exécution, la variable globale y contient le lexème couramment lu.
- *lire* met à jour la variable y au lexème suivant du mot m.
- $ana(t : T)$ donnée en Algorithme 2 prend en argument un terminal de la grammaire et le compare au lexème courant. Cette fonction sert à vérifier que l'entrée est correcte et à passer au lexème suivant.

Algorithme 1 analyseLLRecursive(m)

Entrées: m le mot à analyser suivi du marqueur de fin \$
Sorties: t l'arbre abstrait de m si $m \in L(\mathcal{G})$

1: lire {– on se positionne sur le premier lexème de m}
2: $t := anaE()$ {– procédure correspondant à l'axiome E de la grammaire $\mathcal{G}$}
3: **si** y == "\$" **alors**
4: **retourner** t
5: **sinon**
6: exception : "$m \notin L(\mathcal{G})$"
7: **finsi**

Algorithme 2 $ana(t : T)$

Entrées: t un terminal de la grammaire
y lexème en cours de lecture (variable globale)
1: **si** $t == y$ **alors**
2: lire
3: **sinon**
4: exception "ana : lexème non attendu"
5: **finsi**

À chaque non-terminal A de la grammaire $\mathcal{G}$, on associe une fonction $anaA$ retournant un arbre abstrait. Pour exemple, on donne anaE et anaH :

```
anaE() = r : arbreAbstrait
  r := anaH(anaF())

anaH (debut : arbreAbstrait) = r : arbreAbstrait
  cas  y == ``$''  : r := debut
       y == ``)''  : r := debut
       y == ``-''  : ana(``-''); r := anaH(Moins(debut, anaF()))
       y == ``+''  : ana(``+''); r := anaH(Plus(debut, anaF()))
       autre cas : exception ``anaH, $ ou ) - ou + attendu''
  fincas
```

En utilisant les symboles directeurs obtenus par l'analyse LL(1) de la grammaire $\mathcal{G}'$, déterminer les fonctions $anaF$ et $anaG$ correspondant au non-terminaux F et G.

Question 5

Exécuter l'analyseur pour les mots m_i $(1 \leq i \leq 6)$ de la question 2.

Exercice 2

On rappelle la grammaire suivante définie dans le projet "Structures de Données (1A)"

$$\begin{array}{lcl} \textit{Règle} & \rightarrow & \textit{TermFonc} = \textit{TermeFonc} \\ \textit{TermeFonc} & \rightarrow & \textit{idf} \mid \textit{idf}\ '('\ \textit{ListeArguments}\ ')' \\ \textit{ListeArguments} & \rightarrow & \varepsilon \mid \textit{TermeFonc} \mid \textit{TermeFonc}\ ','\ \textit{ListeArguments} \end{array}$$

Réfléchir à une analyse LL(1) de cette grammaire en ayant comme objectif de définir un analyseur syntaxique par descente récursive, permettant de générer les arbres abstraits des termes et des règles de réécriture.

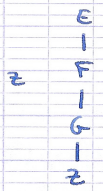
TD 11

MaD

1

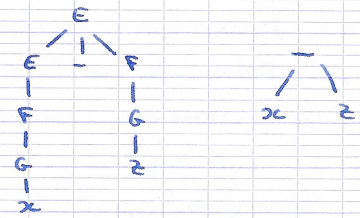
Exercice 1 :

Question 1) $m_1 = z$

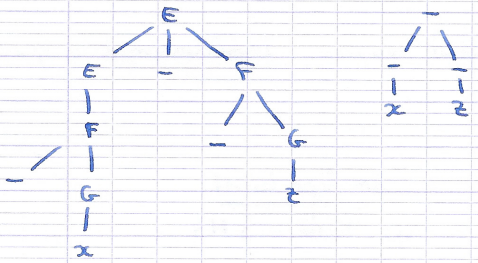


$m_2 = -x$

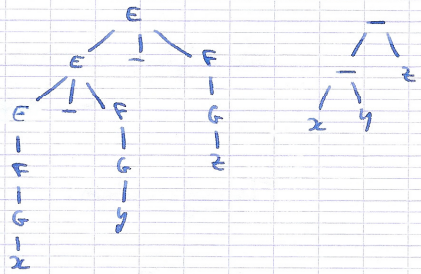
$m_3 = x - z$



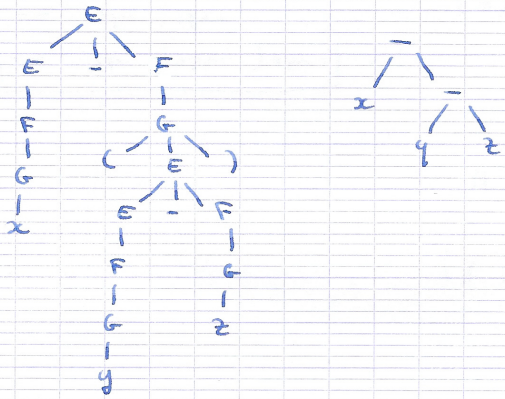
$m_4 = -x - -z$



$m_5 = x - y - z$



$m_6 = x - (y - z)$



Question 2 :

m1 : VarProp ("z")
m2 : MoinsUnaire (VarProp ("x"))
m3 : Moins (VarProp("x"), VarProp("y")).
m4 : Moins (MoinsUnaire(VarProp("x")), MoinsUnaire (VarProp ("z"))).
m5 : Moins (Moins (VarProp("x"), VarProp ("y")), VarProp("z"))
m6 : Moins (VarProp ("x"), Moins (VarProp("y"), VarProp ("z"))).

Question 3 :

Les règles $E \to E+F$ | $E \to E-F$ sont récursives immédiatement à gauche

Transformation :

$E \to FH$
$H \to \varepsilon \mid +FH \mid -FH$
$F \to -F \mid G$
$G \to x \mid y \mid z \mid (E)$.

Analyse LL(1)

① Vide = { H }

② Premier :

E	F	G	H
∅	∅	∅	∅
∅	-	xyz(	+-
-	-xyz(	xyz(	+-
-xyz(	-xyz(	xyz(	+-

③ Suivant :

E	F	G	H
$	-	-	-
$)	+-$)	+-$)	$)

TD 11
MaD
2

Symboles directeurs :

$SD(E \to FH) = \{-, x, y, z, (\}$

$SD(H \to \varepsilon) = \{\#,)\}$
$SD(H \to +FH) = \{+\}$
$SD(H \to -FH) = \{-\}$

tous distincts
donc ok pour dire LL(1)

$SD(F \to -F) = \{-\}$
$SD(F \to G) = \{x, y, z, (\}$

distincts
$\Rightarrow$ LL(1)

$SD(G \to x) = \{x\}$
$SD(G \to y) = \{y\}$
$SD(G \to z) = \{z\}$
$SD(G \to (E)) = \{(\}$

tous distincts
$\Rightarrow$ LL(1).

Question 4 :

anaE () = r : arbreAbstrait
r := anaH(anaF()).

anaF() = r : arbreAbstrait
cas y == "-" : ana("-") : r := MoinsUnaire (anaF()).
autre cas : r = ana G().

anaG() = r : arbreAbstrait
cas y == "x" : ana("x") : r := VarProp ("x").
cas ——— "y" ——— ("y").
cas ——— "z" ——— ("z").