Construction d'analyseurs syntaxiques
TL2 Ensimag 1A 2022-2023

Chapitre 3
Introduction à l'analyse LL(1) sur un exemple

Xavier.Nicollin@grenoble-inp.fr

thanks Sylvain Boulmé *et* Lionel Rieg

Objectifs du chapitre

On cherche à écrire des analyseurs spécifiés à l'aide d'une BNF non-ambiguë attribuée.

Au chapitre suivant, on va définir une classe limitée de BNF, appelée LL(1), ayant de bonnes propriétés pour cet objectif :

- garantit non-ambiguïté, avec un algorithme pour *décider* si une BNF est LL(1)
- des analyseurs efficaces
- un algo simple pour traduire la BNF en programme !

Ici Exercices pour comprendre l'analyse LL(1) sur un exemple simple avec le vocabulaire terminal $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$

cf. corrigé complet dans fichier `chap3.py` sur Chamilo

Exemple illustratif

Soit G_1 la BNF attribuée d'axiome $S{\uparrow}\{a, b, c\}^*$ définie par

(1)	$S{\uparrow}w ::=$	$O{\uparrow}w_1$ c	$w := w_1.a$
(2)	$O{\uparrow}w ::=$	$P{\uparrow}w_1$	$w := w_1.c$
(3)	$\mid$	ε	$w := \varepsilon$
(4)	$P{\uparrow}w ::=$	b	$w := \varepsilon$
(5)		a $P{\uparrow}w_1$ c $P{\uparrow}w_2$	$w := b.w_1.c.a.w_2$

G_1 est "clairement" non ambiguë

G_2 : même BNF sauf (1) remplacée pa "$S \to O$ a"
G_2 est aussi non ambiguë

Une idée de l'analyse LL(1)

LL(1) pour "Left-to-right reading, Leftmost derivation, 1 look-ahead"

Principe Arbres d'analyses construits en *dérivation gauche* (en partant de la racine), en lisant un **seul** terminal "d'avance" : le *prochain* terminal "*non consommé*" – dit de *pré-vision* (ou *look-ahead*) – doit suffire à "*sélectionner*" la règle à appliquer.

Exo 1 Soit $u = \texttt{abcb}$. Appliquer le principe ci-dessus pour construire arbres d'analyses de G_1 (avec attributs) sur
1) c 2) bc 3) u.c 4) a.u.bc 5) a.u.cbc 6) abc.u.c

Directeur LL(1) d'une règle = l'ensemble des terminaux qui peuvent sélectionner cette règle.

Exo 2 Pour chaque règle de G_1, indiquer son directeur LL(1).

Exo 3 Sur G_2 construire arbres d'analyses des 3 mots suivants

ba a abcba

Directeurs LL(1) de G_2 ? Pourquoi pas d'analyse LL(1) pour G_2 ?

Vers une implémentation de G_1 en langage Python

On suppose défini une énumération de terminaux

```
TOKENS = tuple(range(4))
a, b, c, END = TOKENS    # END = token spécial de fin
```

et aussi (voir chap3.py pour les détails).

```
current = END # variable globale de pré-vision

def init_parser(stream): # démarre lecture des tokens
                         # dans 'stream'
                         # et met 'current' sur 1er token

def parse_token(token): # lève 'Error' si 'current!=token'
                        # puis fait avancer 'current'
```

NB current sera uniquement modifié par
init_parser et parse_token

Spécification de l'analyseur LL(1) de G_1

Spécification Pour chaque $X \in \{S, O, P\}$, une fonction `parse_`X consomme *récursivement* le ***plus long préfixe*** (de la suite de tokens non encore *lus*) dérivable de X et **retourne** le w (de type `str`) spéficié par l'équation $X \uparrow w$.
NB L'exécution s'arrête sur `Error` si pas de préfixe reconnu !

D'où la procédure principale qui vérifie que la suite de tokens dans `stream` dérive de G_1 et synthétise le w correspondant :

```
1  def parse(stream):
2      init_parser(stream)
3      w=parse_S()
4      parse_token(END)
5      return w
```

Exo 4 Quel est le rôle de la ligne 4 ?
En supposant qu'on l'ait oubliée, donner un exemple de mot accepté par `parse` mais rejeté par G_1.

Un premier analyseur LL(1)

Exo 5 Implémenter progressivement `parse_S`, `parse_O` et `parse_P` en 3 étapes :

1) Se contenter d'accepter ou rejeter la suite de tokens sans s'occuper de retourner w.

2) Montrer que l'arbre des appels récursifs correspond à l'arbre d'analyse. A essayer sur "`aabcbcbc`"

3) Utiliser cette idée pour affecter correctement l'attribut w.

NB Si nombreuses règles de même membre gauche X avec de "gros" directeurs, alors sélection de règle dans `parse_`X codée par des "`if ... elif ... else: ...`" très inefficace :
coût de chaque sélection de règle en $\Theta(|V_T|)$ dans le pire cas !

Sélection de règle LL(1) à coût constant

Éclater `parse_`X avec une fonction par règle
+ un tableau X`_RULES` qui choisit la règle en fonction de `current`.

```
def X_r1():  # règle 1 de membre gauche X
    ...
def X_rn():  # règle n de membre gauche X

X_RULES = mk_rules([(dir_r1, X_r1), ..., (dir_rn, X_rn)],
                   X_r1)
def parse_X():
    return X_RULES[current]()
```

où chaque "`dir_r`i" = directeur LL(1) de la règle "i" de X
codé sous forme d'une liste de tokens.

avec un `mk_rules` utilisable dans tout analyseur LL(1) :

```
1 def mk_rules(directed_rules, default_rule):
2     rules = [default_rule]*len(TOKENS)
3     for director, rule in directed_rules:
4         for token in director:
5             rules[token] = rule
6     return rules
```

Exo 6 Redéfinir `parse_O` et `parse_P` de G_1 en utilisant ça...