Analyse de Données Structurées - Cours 5

Ralf Treinen

Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes

treinen@pps.univ-paris-diderot.fr

18 février 2015

L'objectif

- L'analyse syntaxique a deux objectifs :
  - détecter si le texte lu est correct (est dans le langage engendré par la grammaire) ;
  - le cas échéant, construire un arbre de syntaxe abstraite.
- Pour l'instant, nous continuons à étudier le premier problème : reconnaissance des textes d'entrée correctes.

L'approche suivie pour l'analyse lexicale

- Vu pour l'analyse lexicale :
  - Le découpage de l'entrée en jetons est spécifiée par des expressions régulières.
  - On sait traduire les expressions régulières en automates finis. Ces automates s'exécutent facilement et de façon efficace.
  - Générateurs d'analyse lexicale : traduction automatique de la spécification en code efficace (qui, lui, contient un automate)

La bonne approche pour l'analyse syntaxique ?

- Peut on suivre la même technique pour l'analyse syntaxique ?
- Il existe un modèle d'automates qui correspond aux grammaires algébriques, dans le même sens que les automates finis correspondent aux expressions régulières : les *automates à pile*.
- On obtient, de la traduction d'une grammaire algébrique, un automate à pile *non-déterministe*.
- En général, on ne peut *pas* rendre cet automate déterministe.

Déterminiser un automate fini

- À chaque moment de l'exécution d'un automate fini, sa configuration consiste en :
  - l'information où on est dans la lecture du mot d'entrée, et
  - l'état de l'automate (nombre fini).
- L'ensemble des configurations possibles dans lesquelles l'automate peut se trouver après lecture d'un mot donné est borné (par le nombre d'états de l'automate).
- Construction d'un automate déterministe de taille exponentielle.

Déterminiser un automate à pile ?

- La configuration d'un automate à pile est plus riche :
  - l'information où on est dans la lecture du mot d'entrée,
  - l'état de l'automate,
  - et le *contenu de la pile*.
- La taille de la pile est *non-bornée*.
- Le nombre des configurations possibles dans lesquelles l'automate peut se trouver après lecture d'un mot n'est plus borné !

Plusieurs techniques pour l'analyse syntaxique

- Analyse descendante : construction (virtuelle) de l'arbre de dérivation, à partir de l'axiome aux feuilles.
  Ordre de construction : parcours préfixe de l'arbre.
  C'est l'approche présentée dans ce cours.
- Analyse ascendante : construction (virtuelle) d'un arbre de dérivation à partir des feuilles jusqu'à l'axiome. Plus complexes à maîtriser, nécessite des connaissances des automates à pile. Voir le cours de Compilation au M1.

Construction (virtuelle) d'un arbre de dérivation

- Dans la construction d'un arbre de dérivation (ou, d'une dérivation), il y a à chaque moment deux choix à faire :
  - du non-terminal qu'on va remplacer à l'aide d'une règle de la grammaire,
  - une fois le non-terminal choisi, de la règle parmi ceux qui ont ce non-terminal sur le côté gauche.
- Nous avons vu la semaine dernière que le premier choix n'est pas essentiel : on peut imposer une stratégie comment choisir le non-terminal à remplacer (par ex., celui qui est le plus à gauche).

Exploration complète de l'espace de recherche ?

- Une façon de réaliser une analyse syntaxique est maintenant d'essayer simplement toutes les possibilités de choisir des règles.
- Cela donner lieu à un algorithme non-déterministe :
  - soit par *retour-en-arrière* (angl. : backtracking)
  - soit par *programmation dynamique*
- Approche complète : on est sûr de trouver un arbre de dérivation si le mot est dans le langage ☺
- Problème : efficacité ☹
- On cherche des solutions efficaces, éventuellement en imposant des restrictions aux grammaires qu'on peut traiter.

Comment obtenir une solution efficace ?

- Il faut maîtriser le choix de la règle de la grammaire par laquelle on va remplacer un non-terminal.
- On ne peut pas demander qu'il y ait une seule règle par non-terminal (car dans ce cas la grammaire est complètement triviale).
- Sur quoi baser le choix de la règle ?
- Sur la suite du mot pour lequel on cherche construire l'arbre de dérivation !

Exemple

- Grammaire $G = (V_T, V_N, S, R)$ où
- $V_T = \{\texttt{i}, \texttt{+}, \texttt{[}, \texttt{]}\}$
- $V_N = \{\mathrm{S}\}$
- $S = \mathrm{S}$
- R consiste en les règles suivantes :

$$\mathrm{S} \rightarrow i \quad (1)$$
$$\mathrm{S} \rightarrow \texttt{[S+S]} \quad (2)$$

- $\mathcal{L}(G)$: expressions complètement parenthésées, construites avec la constante `i` et l'opérateur binaire `+`.

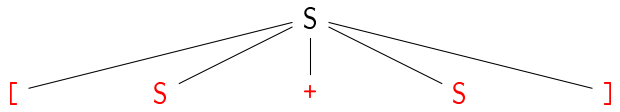
Construction d'un arbre de dérivation (1)

S

[	[	i	+	i	]	+	[	i	+	i	]	]

Choisir règle (2) : c'est la seule qui peut produire à partir de S un mot qui commence sur `[`.

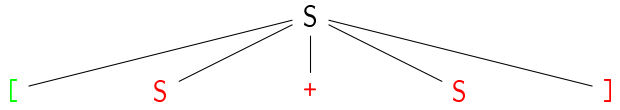
Construction d'un arbre de dérivation (2)



[	[	i	+	i	]	+	[	i	+	i	]	]

Le premier non-terminal du mot des feuilles est [.

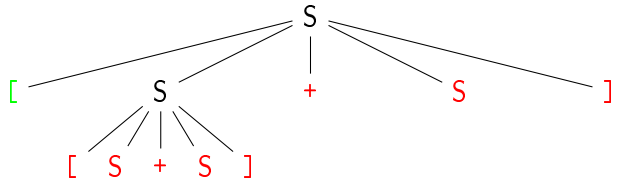
Construction d'un arbre de dérivation (4)



[	[	i	+	i	]	+	[	i	+	i	]	]

Choisir règle (2) : c'est la seule qui peut produire à partir de S un mot qui commence sur [.

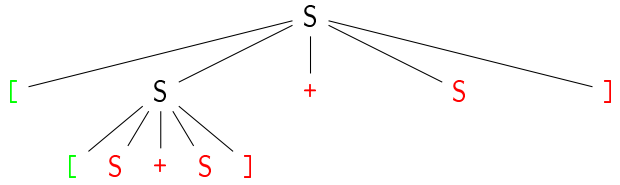
Construction d'un arbre de dérivation (5)



[	[	i	+	i	]	+	[	i	+	i	]	]

Le suivant non-terminal du mot des feuilles est [.

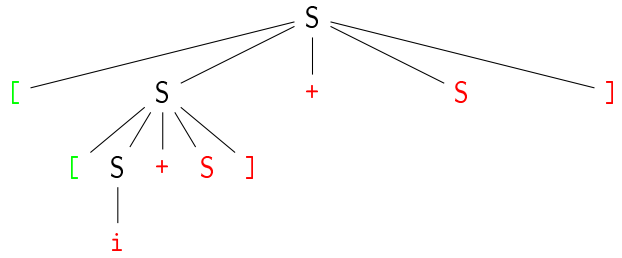
Construction d'un arbre de dérivation (6)



[	[	i	+	i	]	+	[	i	+	i	]	]

Choisir règle (1) : c'est la seule qui peut produire à partir de S un mot qui commence sur i.

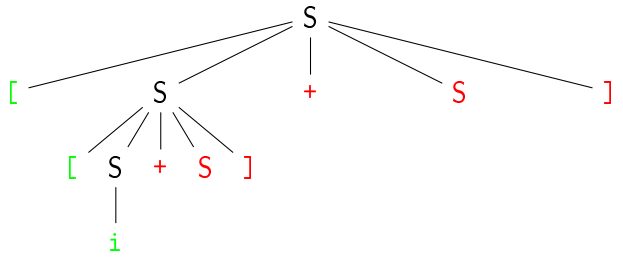
Construction d'un arbre de dérivation (7)



[	[	i	+	i	]	+	[	i	+	i	]	]

Le suivant non-terminal du mot des feuilles est `i`.

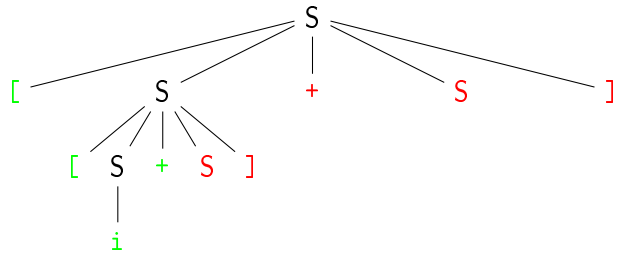
Construction d'un arbre de dérivation (8)



[	[	i	+	i	]	+	[	i	+	i	]	]

Le suivant non-terminal du mot des feuilles est `+`.

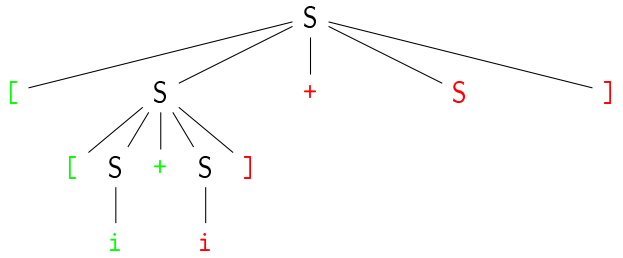
Construction d'un arbre de dérivation (9)



[	[	i	+	i	]	+	[	i	+	i	]	]

Choisir règle (1) : c'est la seule qui peut produire à partir de S un mot qui commence sur `i`.

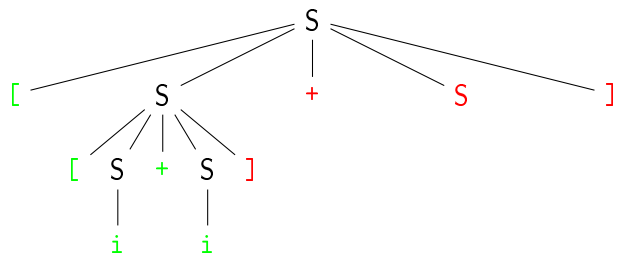
Construction d'un arbre de dérivation (10)



[	[	i	+	i	]	+	[	i	+	i	]	]

Le suivant non-terminal du mot des feuilles est `i`.

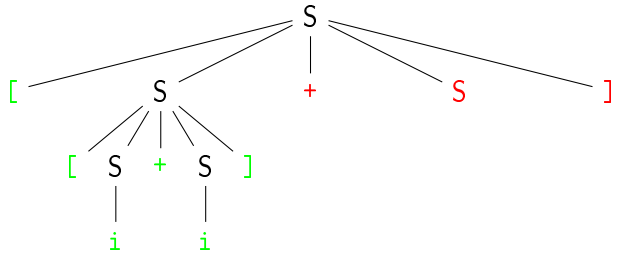
Construction d'un arbre de dérivation (11)



[	[	i	+	i	]	+	[	i	+	i	]	]

Le suivant non-terminal du mot des feuilles est].

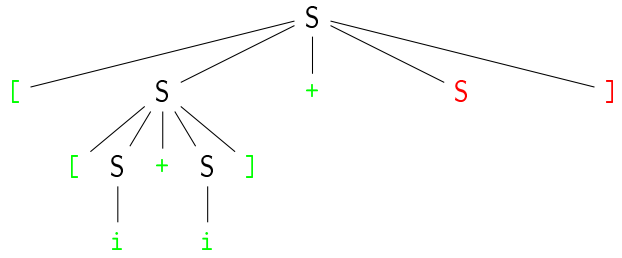
Construction d'un arbre de dérivation (12)



[	[	i	+	i	]	+	[	i	+	i	]	]

Seulement la règle (2) peut produire à partir de S un mot qui commence sur +.

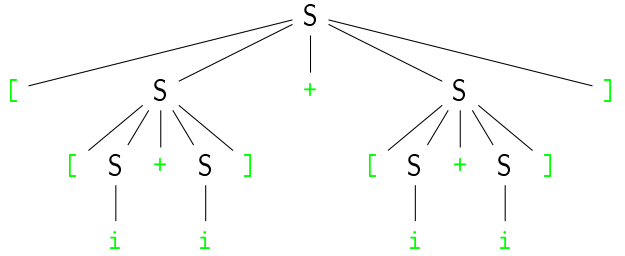
Construction d'un arbre de dérivation (13)



[	[	i	+	i	]	+	[	i	+	i	]	]

Choisir règle (2) : c'est la seule qui peut produire à partir de S un mot qui commence sur [.

etc. etc.



[	[	i	+	i	]	+	[	i	+	i	]	]

Construction terminée !

Ce qu'on a vu sur l'exemple :

- Il y a deux types d'actions :
  - consommer en parallèle un non-terminal du préfixe du mot des feuilles déjà construit, et le même symbole de l'entrée ;
  - ajouter des fils à une feuille de l'arbre de dérivation partiel.
- Pour choisir la règle de la grammaire, on regarde en avant quel est le symbole suivant de l'entrée que nous aurions à consommer (lookahead).

Grammaires LL(1)

En fait, l'algorithme que nous avons vu sur l'exemple appartient à la classe *LL(1)* :

- le premier *L* indique qu'on parcourt l'entrée de la gauche (angl. : left) à la droite ;
- le deuxième *L* indique qu'on construit une dérivation gauche (angl. : left), c.-à-d. un arbre de dérivation dans un ordre préfixe ;
- le nombre *1* indique que nous utilisons la connaissance de 1 caractère dans la partie de l'entrée qui reste à consommer, pour déterminer la règle à appliquer (lookahead=1).

Notation : $w : k$

Définition

Soit $w \in \Sigma^*$ un mot, et $k \in \mathbb{N}$. On définit

- si $|w| \leq k$ alors $w : k = w$
- si $|w| > k$ alors $w : k = x$ tel que $w = xy$ et $|x| = k$

Explication

- $w : k$ est le préfixe de longueur k du mot w, ou le mot w entier si w est plus court que k.
- $abcdefg : 3 = abc$
- $abcd : 7 = abcd$

Définition LL(k)

Définition

Soit $G = (V_T, V_N, S, R)$ une grammaire algébrique, $k \in \mathbb{N}$. G est dite *LL(k)* ssi

- S'il existe deux dérivations gauches

$$S \to^* uY\alpha \to u\beta\alpha \to^* ux$$
$$S \to^* uY\alpha \to u\gamma\alpha \to^* uy$$

  où $Y \in V_N$, $u, x, y \in V_T^*$, $\alpha, \beta, \gamma \in (V_N \cup V_T)^*$, avec $x : k = y : k$
- alors $\beta = \gamma$.

Explication de la définition de LL(k)

- On a déjà consommé le mot de terminaux u.
- Le non-terminal le plus à gauche à réécrire est maintenant Y.
- Dans les deux cas considérés, le mot d'entrée continu une fois par le mot x, l'autre fois par le mot y.
- En regardant les k premiers caractères de la suite du mot d'entrée, on peut maintenant décider comment réécrire le non-terminal y.

Un premier critère simple pour être LL(1)

Lemme

Si pour tout non-terminal, les côtés droites de toutes les règles pour ce non-terminal commencent par des terminaux différents, alors la grammaire est LL(1).

Exemple

La grammaire de l'exemple précédent :

$$\begin{aligned} \mathrm{S} &\to i \\ \mathrm{S} &\to [\mathrm{S}+\mathrm{S}] \end{aligned}$$

satisfait le critère, et est donc LL(1).

Un meilleur critère pour être LL(1) ?

Problème

Le critère du transparent précédent est un peu trop restrictif car il ne permet pas des règles où le côté droite commence par un non-terminal :

$$\begin{aligned} \mathrm{A} &\to \mathrm{BA} \mid \mathrm{B} && (3) \\ \mathrm{B} &\to \ldots && (4) \end{aligned}$$

La fonction FIRST_k

Définition

Soit $G = (V_T, V_N, S, R)$ une grammaire, et $k \in \mathbb{N}$. Nous définissons une fonction

$$\mathrm{FIRST}_k : (V_T \cup V_N)^* \to 2^{V_T^*}$$

par

$$\mathrm{FIRST}_k(\alpha) = \{w : k \mid w \in V_T^*, \alpha \to^* w\}$$

Explication

$\mathrm{FIRST}_k(\alpha)$ est l'ensemble des préfixes de longueur k des mots terminaux qu'on peut obtenir à partir α.

Un meilleur critère pour être LL(1)

Lemme

Soit $G = (V_T, V_N, S, R)$ une grammaire *sans productions de la forme* $N \to \epsilon$. G est LL(1) si est seulement si pour toutes règles différentes :

$$\begin{array}{rcl} \mathsf{N} & \to & \alpha \\ \mathsf{N} & \to & \beta \end{array}$$

on a que $\mathrm{FIRST}_1(\alpha) \cap \mathrm{FIRST}_1(\beta) = \emptyset$.

Exemple

Toujours sur le même exemple :

$$\begin{array}{rcl} \mathrm{FIRST}_1(\mathtt{i}) & = & \{\mathtt{i}\} \\ \mathrm{FIRST}_1(\mathtt{[S+S]}) & = & \{\mathtt{[}\} \end{array}$$

Calcul de FIRST_1

- Grammaire $G = (V_T, V_N, S, R)$. *Hypothèse : aucune production* $N \to \epsilon$.
- On calcule $Fi(\alpha)$ pour tout côté droite de R.
- $Fi(a\alpha) = \{a\}$ pour tout $a \in V_T$
  $Fi(N\alpha) = \emptyset$ pour tout $N \in V_N$
- Tant qu'il existe une règle $N \to \beta$ telle que $Fi(\beta) \not\subseteq Fi(N\alpha)$:

$$Fi(N\alpha) = Fi(N\alpha) \cup Fi(\beta)$$

- Pour tout α : $\mathrm{FIRST}_1(\alpha)$ est la valeur finale de $Fi(\alpha)$.

Exemple

- Grammaire $G = (\{\mathsf{a}, (,), +\}, \{\mathsf{F}, \mathsf{S}\}, \mathsf{S}, R)$ où R est

$$\begin{array}{rcl} \mathsf{F} & \to & \mathsf{a} \\ \mathsf{S} & \to & \mathsf{(F+S)} \\ \mathsf{S} & \to & \mathsf{F} \end{array}$$

- Initialisation :

$$\begin{array}{rcl} Fi(\mathsf{a}) & = & \{\mathsf{a}\} \\ Fi(\mathsf{(F+S)} & = & \{(\} \\ Fi(\mathsf{F}) & = & \emptyset \end{array}$$

- Complétion : On a une règle $\mathsf{F} \to \mathsf{a}$, mais $Fi(\mathsf{a})\neg \subseteq Fi(\mathsf{F})$.
- On augmente : $Fi(\mathsf{F}) = \{\mathsf{a}\}$

Exemple (2)

- Grammaire $G = (\{\mathsf{a}, (,), +\}, \{\mathsf{F}, \mathsf{S}\}, \mathsf{S}, R)$ où R est

$$\begin{array}{rcl} \mathsf{F} & \to & \mathsf{a} \\ \mathsf{S} & \to & \mathsf{(F+S)} \\ \mathsf{S} & \to & \mathsf{F} \end{array}$$

- On obtient donc :

$$\begin{array}{rcl} \mathrm{FIRST}_1(\mathsf{a}) & = & \{\mathsf{a}\} \\ \mathrm{FIRST}_1(\mathsf{(F+S)}) & = & \{(\} \\ \mathrm{FIRST}_1(\mathsf{F}) & = & \{\mathsf{a}\} \end{array}$$

Structure du code

- Nous avons besoin de regarder le symbole suivant dans le flot d'entrée sans de le consommer : classe `LookAhead1Reader`.
- L'analyseur syntaxique contient une méthode `term (char c)` qui consomme un symbole c du flot d'entrée.
- L'analyseur syntaxique contient une méthode `nonterm_N()` pour chaque non-terminal N. Cette méthode consomme un mot du flot d'entrée qui est engendré par N.
  L'implémentation de ces méthodes utilise FIRST_1 pour déterminer la règle de la grammaire à appliquer.

Fichier `LookAhead1Reader.java` I

```
import java.io.*;

class LookAhead1Reader extends PushbackReader {
    /* Reader class with a lookahead of one character */

    public LookAhead1Reader(Reader r) {
        super(r,1);
    }

    public boolean check(char c)
        throws IOException {
        /* check whether c is the first character */
        int lastread=0;
        lastread=this.read();
        this.unread(lastread);
        return (lastread == c);
    }
```


Fichier `LookAhead1Reader.java` II

```
    public void eat(char expected)
        throws ReadException, IOException {
        /* consumes c from the stream, exception    */
        /* when the contents does not start on c.   */
        char found=(char)this.read();
        if (found != expected) {
            throw new ReadException(expected,found);
        }
    }
}
```


Fichier `Parser.java` I

```
import java.io.*;

/* simple LL(1) parser for the grammar :    */
/* F -> a     S -> F     S -> (F+S)         */
class Parser {

    protected LookAhead1Reader reader;

    public Parser(LookAhead1Reader r) {
        reader=r;
    }
```

Fichier Parser.java II

```
    public void term (char c) throws IOException, ReadException {
        /* consume the character c */
        reader.eat(c);
    }

    public void nonterm_S()
        throws IOException, IllegalArgumentException,
               ReadException, ParserException {
        /* parse a word generated from nonterminal S */
        if (reader.check('a')) {
            this.nonterm_F();
        } else if (reader.check('(')) {
```


Fichier Parser.java III

```
            this.term('(');
            this.nonterm_F();
            this.term('+');
            this.nonterm_S();
            this.term(')');
        } else {
            throw new ParserException("cannot␣reduce␣S");
        }
    }

    public void nonterm_F() throws IOException, ReadException {
        /* parse a word generated from nonterminal F */
```


Fichier Parser.java IV

```
        this.term('a');
    }
}
```


Fichier Test.java

```
import java.io.*;

class Test {

    public static void main(String[] args) throws Exception {
        File input = new File(args[0]);
        Reader reader = new FileReader(input);
        LookAhead1Reader r = new LookAhead1Reader(reader);
        Parser p = new Parser(r);
        p.nonterm_S();
    }
}
```