Analyse de Données Structurées - Cours 7

Ralf Treinen

Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes

treinen@pps.univ-paris-diderot.fr

18 mars 2015

Vu au dernier cours

- Calcul d'un point fixe par `do ... until X fix`
- Définition : $Fi(N) = \{a \in V_T \mid N \rightarrow^* a\alpha\}$
- On a que $Fi(N) = FIRST_1(N) \setminus \{\epsilon\}$
- Algorithme pour le calcul de $Fi(N)$ en présence de règles $M \rightarrow \epsilon$
- Reste à faire : calcul de $FIRST_1(\alpha)$ pour $\alpha \in (V_T \cup V_N)^*$

Rappel : Calcul de *Fi*

Algorithme

Donnée une grammaire $G = (V_T, V_N, S, R)$.

pour tout $N \in V_N$:
  Fi(N) = $\{a \in V_T \mid (N \rightarrow N_1 \ldots N_n a\alpha) \in R, N_1, \ldots, N_n \in \text{EPS}\}$
do
  pour tout $(N \rightarrow N_1 \ldots N_n M\alpha) \in R$
  tel que $M \in V_N, N_1, \ldots, N_n \in \text{EPS}$:
    Fi(N) = Fi(N) ∪ Fi(M)
until Fi fix

Exemple

Grammaire

A	→	$\epsilon \mid a$	B	→	$\epsilon \mid$ Ab
C	→	ABc	D	→	BCd

Non-terminaux qui peuvent produire ϵ

$EPS = \{A, B\}$

Calcul

	Initial	*Iter*1	*Iter*2
A	{a}	{a}	{a}
B	{b}	{a, b}	{a, b}
C	{c}	{a, b, c}	{a, b, c}
D	∅	{b, c}	{a, b, c}

Calcul de $FIRST_1$ dans le cas général

Pour tout N $\in V_N$:

$$FIRST_i(\mathrm{N}) = \begin{cases} Fi(N) \cup \{\epsilon\} & \text{si N} \in EPS \\ Fi(N) & \text{sinon} \end{cases}$$

Calcul de $FIRST_1$ sur l'exemple

$$\begin{array}{lcllcl} \mathrm{E} & \to & \mathrm{T\ E'} & \mathrm{E'} & \to & \epsilon \mid \mathtt{+E} \\ \mathrm{T} & \to & \mathrm{F\ T'} & \mathrm{T'} & \to & \epsilon \mid \mathtt{*T} \\ \mathrm{F} & \to & \mathtt{(E)} \mid \mathtt{i} & \mathrm{S} & \to & \mathrm{E\ EOF} \end{array}$$

$$EPS = \{\mathrm{E'}, \mathrm{T'}\}$$

Calcul

	Fi	$FIRST_1$
S	{i, (}	{i, (}
E	{i, (}	{i, (}
E′	{+}	{+, ϵ}
T	{i, (}	{i, (}
T′	{*}	{*, ϵ}
F	{i, (}	{i, (}

Calcul de $FIRST_1$ dans le cas général

On étend maintenant $FIRST_1$ à des mots de terminaux et non-terminaux :

- $FIRST_1(\epsilon) = \{\epsilon\}$
- pour tout $a \in V_T$: $FIRST_1(a\alpha) = \{a\}$
- pour tout $N \in V_N$:

$$FIRST_1(N\alpha) = \begin{cases} FIRST_1(N) & \text{si } N \notin EPS \\ (FIRST_1(N) \setminus \{\epsilon\}) \cup FIRST_1(\alpha) & \text{si } N \in EPS \end{cases}$$

Calcul de $FIRST_1$ des côtés droites dans l'exemple

$$\begin{aligned} FIRST_1(\mathrm{T\ E'}) &= \{\mathtt{i}, (\} \\ FIRST_1(\epsilon) &= \{\epsilon\} \\ FIRST_1(\mathtt{+\ E}) &= \{+\} \\ FIRST_1(\mathrm{F\ T'}) &= \{\mathtt{i}, (\} \\ FIRST_1(\mathtt{*\ T}) &= \{*\} \\ FIRST_1(\mathtt{(\ E\ )}) &= \{(\} \\ FIRST_1(\mathtt{i}) &= \{\mathtt{i}\} \\ FIRST_1(\mathrm{S\ EOF}) &= \{\mathtt{i}, (\} \end{aligned}$$

Nous avons besoin de plus d'information !

- Le calcul de $FIRST_1$ n'est plus suffisant pour savoir quelle production appliquer !
- Exemple : $E' \to \epsilon \mid +E$
  Si nous voyons + alors il faut utiliser la deuxième alternative pour réécrire E'. Mais quand faut il appliquer la première ?
- Il nous manque une information : quel sont les symboles terminaux qui peuvent *suivre* à un mot produit par un non-terminal ?

La fonction $FOLLOW_k$

Définition

Soit $G = (V_T, V_N, S, R)$ une grammaire, $k \in \mathbb{N}$. La fonction $FOLLOW_k : V_N \to 2^{V_T^*}$ est définie par

$$FOLLOW_k(N) = \{w \mid S \to^* \beta N \gamma, \beta, \gamma \in (V_T \cup V_N)^*, w \in FIRST_k(\gamma)\}$$

Remarques

- $FOLLOW_k(N)$ est l'ensemble de tous les mots de terminaux de longueur k qui peuvent, dans des mots de $\mathcal{L}(G)$, suivre à un mot dérivé de N.

Calcul de $FOLLOW_1$

Algorithme

Donnée une grammaire $G = (V_T, V_N, S, R)$.

for all $N \in V_N$:
  Fo(N)=$\{a \in V_T \mid (M \to \ldots NN_1 \ldots N_k a \ldots) \in R, N_1, \ldots, N_n \in \text{EPS}\}$
for all $(M \to \ldots NN_1 \ldots N_k N' \ldots) \in R$
with $N_1, \ldots, N_k \in$ EPS :
  Fo(N) = $Fo(N) \cup Fi(N')$
do
  for all $(M \to \ldots NN_1 \ldots N_k) \in R$
  with $N_1, \ldots, N_n \in$ EPS :
    Fo(N) = Fo(N) $\cup$ Fo(M)
until Fo fix

Lemme

Si $(S \to N\ \texttt{EOF}) \in R$, alors $\forall N \in V_N : Fo(N) = FOLLOW_1(N)$

Calcul de $FOLLOW_1$ sur l'exemple

$$\begin{array}{llllll} E & \to & T\ E' & E' & \to & \epsilon \mid +E \\ T & \to & F\ T' & T' & \to & \epsilon \mid *T \\ F & \to & (E) \mid \texttt{i} & S & \to & E\ \texttt{EOF} \end{array}$$

$$EPS = \{E', T'\} \quad Fi(E') = \{+\} \quad Fi(T') = \{*\}$$

Calcul

	Initial	*Iter*1	*Iter*2
S	$\emptyset$	$\emptyset$	$\emptyset$
E	$\{\texttt{EOF},)\}$	$\{\texttt{EOF},)\}$	$\{\texttt{EOF},)\}$
E'	$\emptyset$	$\{\texttt{EOF},)\}$	$\{\texttt{EOF},)\}$
T	$\{+\}$	$\{+, \texttt{EOF},)\}$	$\{+, \texttt{EOF},)\}$
T'	$\emptyset$	$\{+\}$	$\{\texttt{EOF},), +\}$
F	$\{*\}$	$\{*, +\}$	$\{*, \texttt{EOF},), +\}$

Finalement (!) : le critère pour être LL(1)

Théorème

La grammaire $G = (V_T, V_N, S, R)$ est LL(1) ssi pour toutes les alternatives $A \to \alpha_1 \mid \ldots \mid \alpha_n$:

- $FIRST_1(\alpha_1), \ldots, FIRST_1(\alpha_n)$ sont disjoints entr'eux ;
- Si $\epsilon \in FIRST_1(\alpha_i)$, alors pour tous $j \neq i$:

$$FIRST_1(\alpha_j) \cap FOLLOW_1(A) = \emptyset$$

Remarque

Condition (1) implique qu'au plus un des ensembles $FIRST_1(\alpha_i)$ contient ϵ.

Le critère sur l'exemple

Non-terminal	Cas 1	$FIRST_1$	Cas 2	$FIRST_1$
E	T E′	{i, (}		
E′	ϵ	{ϵ}	+ E	{+}
T	F T′	{i, (}		
T′	ϵ	{ϵ}	* T	{*}
F	(E)	{(}	i	{i}
S	E EOF	{i, (}		

- $FOLLOW_1(\text{E}') = \{), \texttt{EOF}\}$ disjoint avec {+} ☺
- $FOLLOW_1(\text{T}') = \{\texttt{EOF},), +\}$ disjoint avec {*} ☺
- {(} disjoint avec {i} ☺
- Conclusion : la grammaire est LL(1) !

Comment choisir la règle dans l'analyse syntaxique

Soit $A \to \alpha_1 \mid \ldots \mid \alpha_n$ une alternative. Il y deux cas :

1. Soit, aucun des $FIRST_1(\alpha_i)$ ne contient ϵ :
   comme avant :
   - On choisit la règle $A \to \alpha_i$ quand le symbole suivant est dans $FIRST_1(\alpha_i)$ (ils sont tous disjoints).
   - Erreur si aucun tel i existe
2. Soit α_i avec $\epsilon \in FIRST_i(\alpha_i)$:
   - si le symbole suivant est dans $FIRST_1(\alpha_j)$: choisir $A \to \alpha_j$, pour $1 \leq j \leq n$.
   - si le symbole suivant est dans $FOLLOW_1(A)$: choisir $A \to \alpha_i$.
   - sinon Erreur.

Le code du l'analyseur de syntaxe sur l'exemple I

```
import java.io.*;

/* LL(1) parser for expressions with priorities, */
/* terminated by the symbol '$'                  */
class Parser {

    protected LookAhead1Reader reader;

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

    public void term (char c) throws IOException, ReadException {
        /* consume the character c */
```

Le code du l'analyseur de syntaxe sur l'exemple II

```
        reader.eat(c);
    }

    public void nonterm_S() throws IOException, ReadException {
        /* S -> E $ */
        this.nonterm_E();
        this.term('$');
    }

    public void nonterm_E() throws IOException, ReadException {
        /* E -> T E1 */
        this.nonterm_T();
        this.nonterm_E1();
    }
```


Le code du l'analyseur de syntaxe sur l'exemple III

```
    public void nonterm_E1() throws IOException, ReadException {
        if (reader.check('$') || reader.check(')')) {
            /* E1 -> epsilon */
        } else if (reader.check('+')) {
            /* E1 -> + E */
            this.term('+');
            this.nonterm_E();
        }
    }

    public void nonterm_T() throws IOException, ReadException {
        /* T -> F T1 */
        this.nonterm_F();
        this.nonterm_T1();
    }
```


Le code du l'analyseur de syntaxe sur l'exemple IV

```
    public void nonterm_T1() throws IOException, ReadException {
        if (reader.check('$') || reader.check(')') || reader.check('+'))
            /* T1 -> epsilon */
        } else if (reader.check('*')) {
            /* T1 -> * T */
            this.term('*');
            this.nonterm_T();
        }
    }

    public void nonterm_F() throws IOException, ReadException {
        if (reader.check('(')) {
            /* F -> ( E ) */
            this.term('(');
```


Le code du l'analyseur de syntaxe sur l'exemple V

```
            this.nonterm_E();
            this.term(')');
        } else if (reader.check('i')) {
            /* F -> i */
            this.term('i');
        }
    }
}
```

Syntaxe concrète

- *Syntaxe concrète* : c'est la définition de la forme correcte de l'écriture (représentation textuelle). Elle est normalement définie par des expressions régulières, et une grammaire hors-contexte.
- La vérification qu'un texte d'entrée suit la syntaxe concrète d'un langage est réalisée en coopération par l'analyse lexicale et l'analyse syntaxique.
- Or, normalement on ne veut pas seulement savoir savoir si l'entrée est correcte ou pas, mais aussi faire quelque chose avec, par exemple :
  - Document XML représentant des données géographiques : afficher une carte.
  - Programme écrit dans un langage de programmation : l'exécuter (cas d'un interpréteur), ou engendrer du code exécutable (cas d'un compilateur).

- On va donc construire une représentation de la structure que les analyses lexicales et syntaxiques ont découvertes : c'est la *syntaxe abstraite*.
- Le passage de la syntaxe concrète à la syntaxe abstraite a deux fonctions :
  - Vérifier que l'entrée correspond aux règles de la syntaxe concrète,
  - si c'est le cas, construire une représentation en forme de la syntaxe abstraite.
- Il y a ici souvent une *abstraction* : certains détails de la représentation textuelle (en syntaxe concrète) sont omis dans la syntaxe abstraite car ils ne sont pas pertinents pour la suite.
- On essaye d'abstraire de tout qui n'est pas nécessaire pour la suite : simplifier tant que possible !

Abstraction

- Quels sont les détails de la syntaxe concrète dont on peut faire abstraction : ça dépend de ce qu'on compte faire avec !
- Nous avons déjà vu certaines abstractions faites par l'analyse lexicale, dans le contexte des langages de programmation :
  - les espaces (mais attention aux cas où l'incrémentation des lignes indique la structure du programme)
  - les commentaires (mais attention aux cas où les commentaires sont importantes pour la suite, par exemple si on écrit un *pretty printer* de code source)
- Dans un premier temps on peut utiliser l'arbre de dérivation construit par l'analyse syntaxique comme syntaxe abstraite.

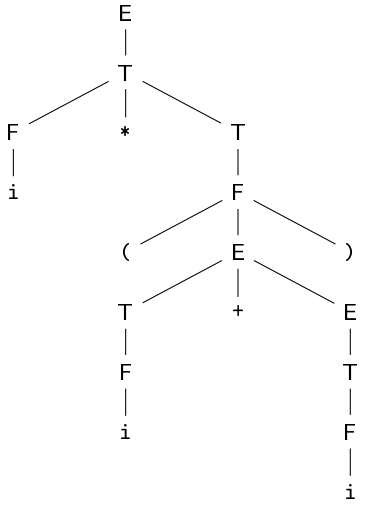
Arbre de dérivation vs. Syntaxe abstraite

- En général, l'arbre de dérivation contient trop d'information.
- Raison : l'arbre de dérivation dit *comment* la structure a été découverte dans le texte d'entrée. Or, c'est seulement la structure elle-même qui est pertinente pour la suite.
- Exemple : grammaire

$$\begin{aligned} \mathsf{E} &\rightarrow \mathsf{E{+}T} \mid \mathsf{T} \\ \mathsf{T} &\rightarrow \mathsf{T{*}F} \mid \mathsf{F} \\ \mathsf{F} &\rightarrow \mathtt{(}\mathsf{E}\mathtt{)} \mid \mathtt{i} \end{aligned}$$

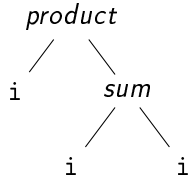
- Arbre de dérivation pour $i * (i + i)$: voir le transparent suivant.

Arbre de dérivation pour $i * (i + i)$

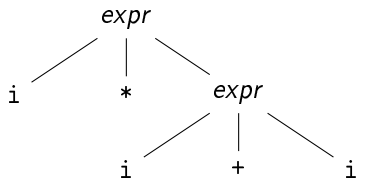


Quel information retenir dans la syntaxe abstraite ?

- La distinction entre E, T et F n'est plus pertinente.
- Les parenthèses ne sont plus pertinentes non plus : elles servent à indiquer la structure dans la syntaxe concrète, mais une fois la structure découverte elle ne servent plus à rien.
- Structure à retenir :



Autre possibilité pour la syntaxe abstraite



- Ici on aura une seule classe, avec l'opérateur comme argument supplémentaire du constructeur.
- Solution à préférer si dans la suite tous les opérateurs sont traités de la même façon.

Définition de la Syntaxe Abstraite sur l'exemple I

```
abstract class Expression {
    public boolean isIdentifier() {
        return false;
    }
    public boolean IsSum() {
        return false;
    }
    public boolean IsProduct() {
        return false;
    }
    abstract void print();

}
```

Définition de la Syntaxe Abstraite sur l'exemple II

```
class Identifier extends Expression {
    public boolean isIdentifier() {
        return true;
    }
    public Identifier () {
    }
    public void print() {
        System.out.print("i");
    }
}

class Sum extends Expression {
    private Expression left;
    private Expression right;
    public boolean isSum() {
```


Définition de la Syntaxe Abstraite sur l'exemple III

```
        return true;
    }
    public Sum(Expression e1, Expression e2) {
        left=e1;
        right=e2;
    }
    public void print() {
        System.out.print("Sum(");
        left.print();
        System.out.print(',');
        right.print();
        System.out.print(")");
    }
}
```


Définition de la Syntaxe Abstraite sur l'exemple IV

```
class Product extends Expression {
    private Expression left;
    private Expression right;
    public boolean isProduct() {
        return true;
    }
    public Product(Expression e1, Expression e2) {
        left=e1;
        right=e2;
    }

    public void print() {
        System.out.print("Product(");
        left.print();
        System.out.print(',');
```


Définition de la Syntaxe Abstraite sur l'exemple V

```
        right.print();
        System.out.print(")");
    }

}
```

Construction de la Syntaxe Abstraite sur l'exemple I

```
import java.io.*;

/* LL(1) parser for expressions with priorities, */
/* terminated by the symbol '$'                  */
class Parser {

    protected LookAhead1Reader reader;

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

    public void term (char c) throws IOException, ReadException {
        /* consume the character c */
```


Construction de la Syntaxe Abstraite sur l'exemple II

```
        reader.eat(c);
    }

    public Expression nonterm_S()
        throws IOException, ReadException, ParserException {
        /* S -> E $ */
        Expression e = this.nonterm_E();
        this.term('$');
        return e;
    }

    public Expression nonterm_E()
        throws IOException, ReadException, ParserException {
        /* E -> T E1 */
        Expression e1 = this.nonterm_T();
```


Construction de la Syntaxe Abstraite sur l'exemple III

```
        Expression e2 = this.nonterm_E1();
        if (e2 == null) {
            return e1;
        } else {
            return new Sum(e1,e2);
        }
    }

    public Expression nonterm_E1()
        throws IOException, ReadException, ParserException {
        if (reader.check('$') || reader.check(')')) {
            /* E1 -> epsilon */
            return null;
        } else if (reader.check('+')) {
            /* E1 -> + E */
```


Construction de la Syntaxe Abstraite sur l'exemple IV

```
            this.term('+');
            return this.nonterm_E();
        } else {
            throw new ParserException("cannot␣reduce␣E'");
        }
    }

    public Expression nonterm_T()
        throws IOException, ReadException, ParserException {
        /* T -> F T1 */
        Expression e1 = this.nonterm_F();
        Expression e2 = this.nonterm_T1();
        if (e2 == null) {
            return e1;
        } else {
```

Construction de la Syntaxe Abstraite sur l'exemple V

```
            return new Product(e1,e2);
        }
    }

    public Expression nonterm_T1()
        throws IOException, ReadException, ParserException {
        if (reader.check('$') || reader.check(')') || reader.check('+'))
            /* T1 -> epsilon */
            return null;
        } else if (reader.check('*')) {
            /* T1 -> * T */
            this.term('*');
            return this.nonterm_T();
        } else {
            throw new ParserException("cannot reduce T'");
```


Construction de la Syntaxe Abstraite sur l'exemple VI

```
        }
    }

    public Expression nonterm_F()
        throws IOException, ReadException,ParserException {
        if (reader.check('(')) {
            /* F -> ( E ) */
            this.term('(');
            Expression e = this.nonterm_E();
            this.term(')');
            return e;
        } else if (reader.check('i')) {
            /* F -> i */
            this.term('i');
            return new Identifier();
```


Construction de la Syntaxe Abstraite sur l'exemple VII

```
        } else {
            throw new ParserException("cannot reduce F");
        }
    }
}
```