Analyse syntaxique

4 octobre 2010

1. Analyse syntaxique
   - Introduction
   - Analyse descendante
   - Analyse ascendante
   - Conflits et Précédences
   - Arbres de syntaxe abstraite
   - Conclusion

Rappels

Une grammaire G est composée de:

- Ensemble de terminaux: T
- Ensemble de non-terminaux: N, symbole initial $S \in N$.
- Ensemble de règles de la forme

$$X ::= x_1 \dots x_n \text{ avec } x_i \in N \cup T$$

Exemple

$T = \{a, b\}$, $N = \{X\}$

$$\begin{array}{ll} X & ::= a\ X\ b \\ X & ::= \epsilon \\ X & ::= X\ X \end{array}$$

Remarques

- Règles récursives $X ::= aXb$
- Règle pour produire le mot vide

Dérivations

du symbole initial à un mot sur l'alphabet terminal

$$X \to XX \to aXbX \to aaXbbX \to aaXbbaXb \to aabbaXb \to aabbab$$

- Dérivation gauche : remplacement du non-terminal le plus à gauche;
- Dérivation droite : remplacement du non-terminal le plus à droite.

Langage reconnu

Une grammaire reconnait un ensemble de mots formés sur les symboles terminaux.
Ce sont les mots obtenus par dérivation à partir du symbole initial.

$$S \rightarrow m_1 \rightarrow \cdots m_n \in T^*$$

On a $m \rightarrow m'$ si:

- m contient un non-terminal X
- La grammaire contient une règle $X ::= x_1 \dots x_n$
- m' est obtenu en remplaçant une occurrence de X par $x_1 \dots x_n$

Arbre de dérivation syntaxique

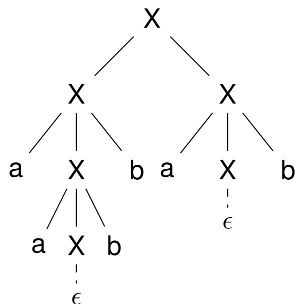
Un mot m est reconnu s'il existe un arbre de dérivation syntaxique.

- les noeuds internes contiennent des symboles non-terminaux
- les feuilles contiennent des symboles terminaux
- la racine est le symbole initial S
- dans un parcours infixe, les feuilles forment le mot m
- si un noeud interne étiqueté X a des sous-arbres $t_1, \ldots, t_n$ de racines $x_1, \ldots, x_n$ alors $X ::= x_1 \ldots x_n$ est une règle de la grammaire.

Remarque

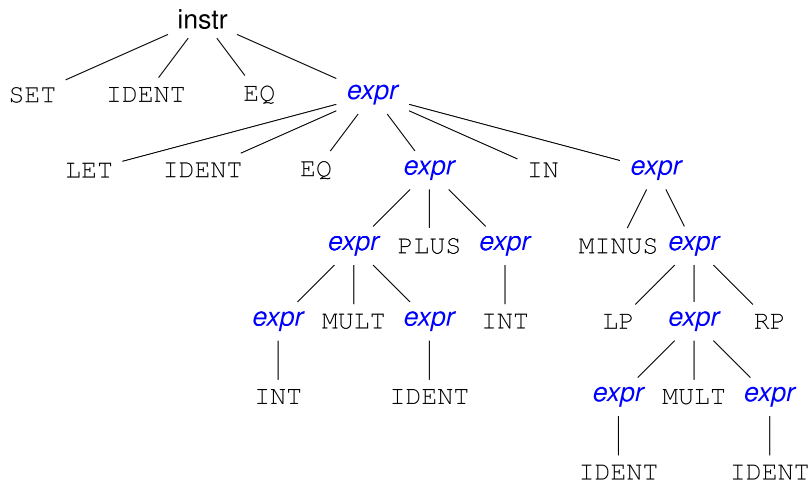
ne pas confondre arbre de dérivation syntaxique (associé à la grammaire) et arbre de syntaxe abstraite associé au langage.

Exemple



Exemple: langage ARITH

```
set xx = let y = 2 * x + 5 in - (y * y)
```



Principe de l'analyse syntaxique

- Entrée : une suite de tokens (le mot à reconnaitre)
- Recherche d'un arbre de dérivation syntaxique pour ce mot
- Reconstruction d'informations à partir de cet arbre (une action associée à chaque règle)

Comment reconstruire l'arbre ?

Sommaire

1 Analyse syntaxique
- Introduction
- Analyse descendante
- Analyse ascendante
- Conflits et Précédences
- Arbres de syntaxe abstraite
- Conclusion

Principe de l'analyse descendante

On essaie de construire l'arbre de dérivation syntaxique en partant de la racine.

- Entrée: un mot m (ou l'indice dans une chaîne globale).
- Une fonction associée à chaque terminal x et non terminal X.
- Suppose qu'étant donné le non-terminal et le mot, on peut décider de la règle à appliquer.
- Pour reconnaître le mot entier, on part de la fonction associée au symbole initial S, on vérifie qu'on a atteint la fin de chaîne.

Reconstruction d'information:

- Construit (implicitement) un arbre de racine X dont les feuilles forment un préfixe de m.
- Renvoie l'information associée au sous-arbre de racine X (par exemple l'arbre de dérivation syntaxique) et le reste du mot m.

Exemple

On suppose le mot *m* à reconnaître déclaré globalement et *n* représente la première lettre non traitée

```
let a n = if m.[n]='a' then (n+1,info_a())
                else raise ParseError
let b n = if m.[n]='b' then (n+1,info_b())
                else raise ParseError
let rec X n = match guess_X n with
     "aXb" -> let (n1,i1) = a n
               in  let (n2,i2) = X n1
               in  let (n3,i3) = b n2
               in (n3,info_X_aXb(i1,i2,i3))
   | "ϵ" -> (n,info_X_ϵ())
   | "XX"  -> let (n1,i1) = X n
               in  let (n2,i2) = X n1
                in (n2,info_X_XX(i1,i2))
```

Limites de l'analyse descendante

- On ne peut pas toujours décider de la règle à appliquer.
- On s'intéresse aux grammaires LL(1) dans lesquelles on peut décider en fonction du premier caractère à lire $m.[n]$.
- Les grammaires récursives gauches ($X ::= Xm$) ne sont pas LL(1).
- On peut parfois les transformer pour obtenir des grammaires LL(1).

Exemple d'élimination de la récursion

Grammaire des expressions arithmétiques sous une forme LL(1)

S	::=	E
E	::=	T E'
E'	::=	+ T E'
E'	::=	ϵ
T	::=	F T'
T'	::=	$\star$ F T'
T'	::=	ϵ
F	::=	`IDENT`
F	::=	`INT`
F	::=	(E)

Analyseur descendant

analyseur (sans reconstruction de valeur)

```
let rec S n = let n1 = E n in
        if n1 <> String.length m then raise ParseError
and E n = let n1 = T n in E' n1
and E' n = if m.[n]='+' then let n1 = T (n+1) in E' n1
           else n
and T n = let n1 = F n in T' n1
and T' n = if m.[n]='*' then let n1 = F (n+1) in T' n1
           else n
and F n =  if m.[n]=IDENT || m.[n]=INT then (n+1)
           else if m.[n]='(' then
               let n1 = E (n+1) in
                 if m.[n1]=')' then n1+1
                 else raise ParseError
```

A propos d'analyse descendante

- Assez simple à programmer sans faire appel à des automates.
  - Calcul de l'ensemble des premiers caractères dérivables à partir du membre droit d'une règle.
  - $$\begin{array}{lll} \text{PREM}(\epsilon) & = \emptyset & \\ \text{PREM}(xm) & = \{x\} & x \in T \\ \text{PREM}(Xm) & = \text{PREM}(X) & X \in N, X \not\longrightarrow^* \epsilon \\ \text{PREM}(Xm) & = \text{PREM}(X) \cup \text{PREM}(m) & X \in N, X \longrightarrow^* \epsilon \\ \text{PREM}(X) & = \cup_{X:=\,m}\text{PREM}(m) & \end{array}$$
- Limitation des grammaires LL(1): grammaires peu naturelles:
  L'arbre de dérivation syntaxique est assez éloigné de l'arbre de syntaxe abstraite.

Sommaire

1. Analyse syntaxique
   - Introduction
   - Analyse descendante
   - Analyse ascendante
   - Conflits et Précédences
   - Arbres de syntaxe abstraite
   - Conclusion

Analyse ascendante

- L'analyse construit l'arbre de dérivation syntaxique en partant des feuilles.
- On garde en mémoire dans une pile une liste de non-terminaux et de terminaux correspondant à la portion d'arbre reconstruite.
- Deux opérations:
  - lecture/shift : on fait passer le terminal du mot à lire vers la pile
  - réduction/reduce : on reconnait sur la pile la partie droite $x_1 \ldots x_n$ d'une règle $X ::= x_1 \ldots x_n$ et on la remplace par X
- Le mot est reconnu si on termine avec le mot vide à lire et le symbole initial sur la pile.

Exemple de dérivation

	pile	entrée	action
1	ϵ	$id + id * id$	lecture
2	id	$+id * id$	reduction $E ::= id$
3	E	$+id * id$	lecture
4	$E+$	$id * id$	lecture
5	$E + id$	$*id$	reduction $E ::= id$
6	$E + E$	$*id$	lecture
7	$E + E*$	id	lecture
8	$E + E * id$	ϵ	reduction $E ::= id$
9	$E + E * E$	ϵ	reduction $E ::= E * E$
10	$E + E$	ϵ	reduction $E ::= E + E$
11	E	ϵ	reduction $S ::= E$
12	S	ϵ	succès

Décider de l'action

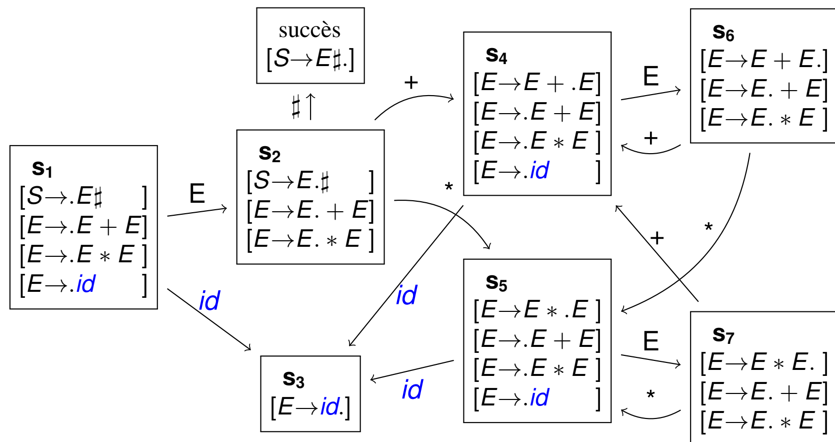
- A chaque état de la pile est associé un état d'un automate fini.
- Une table indique en fonction de l'état courant de la pile et du mot restant à analyser si on doit lire ou réduire.
  - Si lecture le nouvel état dépend de l'état courant et du caractère lu
  - Si réduction par $X ::= x_1 \ldots x_n$ alors le nouvel état dépend de l'état avant la reconnaissance de $x_1 \ldots x_n$ et de X.
- Chaque état correspond aux parties droites de règle qui pourraient être reconnues à ce point de l'analyse.

Exemple d'automate LR(0)

Grammaire :

S	::=	E ♯
E	::=	E + E
E	::=	E * E
E	::=	*id*

Automate :



Retour sur l'exemple

	pile	entrée	action
1	s_1	$id + id * id\sharp$	lecture
2	$s_1.id.s_3$	$+id * id\sharp$	reduction $E ::= id$
3	$s_1.E.s_2$	$+id * id\sharp$	lecture
4	$s_1.E.s_2. + .s_4$	$id * id\sharp$	lecture
5	$s_1.E.s_2. + .s_4.id.s_3$	$*id\sharp$	reduction $E ::= id$
6	$s_1.E.s_2. + .s_4.E.s_6$	$*id\sharp$	lecture
7	$s_1.E.s_2. + .s_4.E.s_6. * .s_5$	$id\sharp$	lecture
8	$s_1.E.s_2. + .s_4.E.s_6. * .s_5.id.s_3$	$\sharp$	reduction $E ::= id$
9	$s_1.E.s_2. + .s_4.E.s_6. * .s_5.E.s_7$	$\sharp$	reduction $E ::= E * E$
10	$s_1.E.s_2. + .s_4.E.s_5$	$\sharp$	reduction $E ::= E + E$
11	$s_1.E.s_2$	$\sharp$	reduction $S ::= E$
12	$s_1.S.\mathrm{succ}$	ϵ	succès

Actions possibles

- Si $s \xrightarrow{a} s'$ avec $a \in T$ alors on peut lire a à partir de s et passer en s'.
- Si s contient un item final $X \rightarrow x_1 \ldots x_n.$ alors on peut faire une réduction par la règle $X ::= x_1 \ldots x_n$.

  On repart de l'état s' avant $x_1 \ldots x_n$ et on applique la transition $s' \xrightarrow{X} s''$.
- Conflits :
  - shift/reduce entre une lecture et une réduction
  - reduce/reduce entre deux réductions
- Les conflits peuvent parfois être résolus en identifiant les terminaux qui peuvent suivre le non-terminal lié à la réduction
- Les conflits peuvent correspondre à des ambiguités dans la grammaire ou bien à des limitations de l'analyse.

Exemple

$$
\begin{array}{ll}
S & ::= X \sharp \\
X & ::= aXb \\
X & ::= aXbaXb \\
X & ::= \epsilon
\end{array}
$$

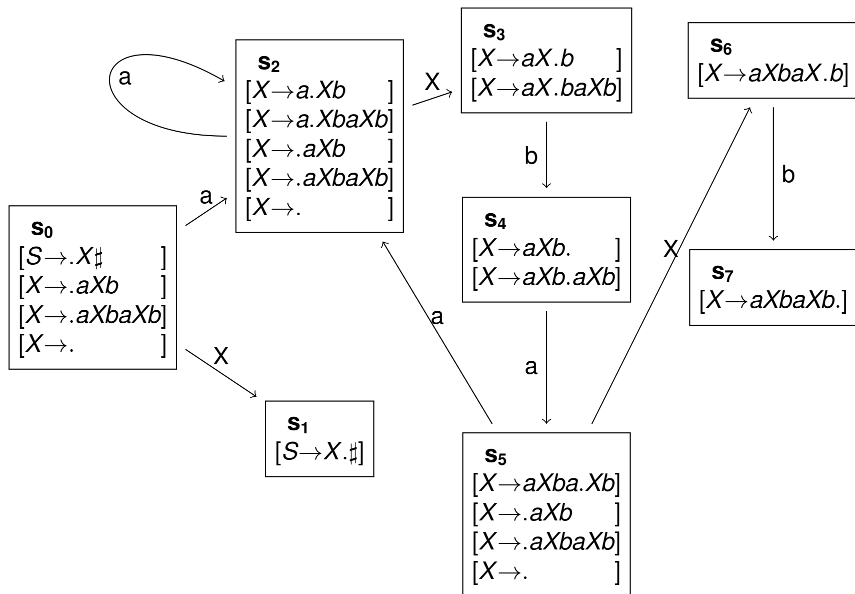
On constate que les seuls terminaux qui suivent la dérivation d'un X sont $\{b, \sharp\}$.

Calcul des suivants d'un non-terminal dans une gammaire:

- $\mathrm{SUIV}(X) = \{a \mid S \longrightarrow^* m_1 X a m_2\}$
- On regarde toutes les occurrences de X dans la partie droite d'une règle $Y ::= m_1 X m_2$:

$$\mathrm{SUIV}(X) = \bigcup_{Y ::= m_1 X m_2} \mathrm{PREM}(m_2) \cup \bigcup_{\substack{Y ::= m_1 X m_2 \\ m_2 \longrightarrow \epsilon}} \mathrm{SUIV}(Y)$$

Construction de l'automate



Sommaire

1. Analyse syntaxique
   - Introduction
   - Analyse descendante
   - Analyse ascendante
   - Conflits et Précédences
   - Arbres de syntaxe abstraite
   - Conclusion

Précédences

- Les conflits reduce-reduce sont à éviter.
- Les conflits shift-reduce peuvent souvent être résolus à l'aide de précédences :
  - La règle a une précédence qui est par défaut celle du terminal le plus à droite mais qui peut être forcée (**%prec** *terminal*)
  - On compare la précédence de la règle *R* et du caractère à lire *c*:

`prec`(R) < `prec`(c)	lecture
`prec`(R) > `prec`(c)	réduction
`prec`(R) = `prec`(c)	associativité gauche : réduction
	associativité droite : lecture

Mise en œuvre ocamlyacc

- `ocamlyacc` indique le nombre et la nature des conflits
- `ocamlyacc -v` *parser.mly* permet d'avoir une trace de l'automate sous-jacent dans un fichier `parser.output`.

Exemple

Grammaire du langage ARITH sans indication de précédence.

```
 1  prog : instrs EOF

 2  instrs : instr
 3         | instrs instr

 4  instr : SET IDENT EQ expr
 5        | PRINT expr

 6  expr : CST
 7       | IDENT
 8       | expr PLUS expr
 9       | expr MINUS expr
10       | expr TIMES expr
11       | expr DIV expr
12       | MINUS expr
13       | LET IDENT EQ expr IN expr
14       | LP expr RP
```

Exemple d'état

```
state 14
	instr : PRINT expr .  (5)
	expr : expr . PLUS expr  (8)
	expr : expr . MINUS expr  (9)
	expr : expr . TIMES expr  (10)
	expr : expr . DIV expr  (11)

	PLUS  shift 21
	MINUS  shift 22
	TIMES  shift 23
	DIV  shift 24
	SET  reduce 5
	PRINT  reduce 5
	EOF  reduce 5
```

Exemple d'état avec conflit

```
29: shift/reduce conflict (shift 21, reduce 9) on PLUS
29: shift/reduce conflict (shift 22, reduce 9) on MINUS
29: shift/reduce conflict (shift 23, reduce 9) on TIMES
29: shift/reduce conflict (shift 24, reduce 9) on DIV
state 29
        expr : expr . PLUS expr  (8)
        expr : expr . MINUS expr  (9)
        expr : expr MINUS expr .  (9)
        expr : expr . TIMES expr  (10)
        expr : expr . DIV expr  (11)

        PLUS  shift 21
        MINUS  shift 22
        TIMES  shift 23
        DIV  shift 24
        SET  reduce 9
        IN  reduce 9
        PRINT  reduce 9
        EOF  reduce 9
        RP  reduce 9
```

Résolution des conflits en ocamlyacc

- reduce/reduce : la première règle est utilisée.
- shift/reduce :
  - utilisation des précédences ou de l'associativité;
  - si la règle et le token ont la même précédence et sont déclarés non associatifs alors c'est une erreur de syntaxe;
  - si pas de règle de précédence alors c'est l'action shift qui est choisie.

Il est conseillé de résoudre explicitement les conflits en modifiant la grammaire et/ou en indiquant les précédences

Syntaxe des fichiers ocamlyacc

```
%{
  header
  (* commentaire *)
%}
  declarations
  /* commentaire */
%%
  rules
  /* commentaire */
%%
  trailer
  (* commentaire *)
```

Exercice

$$
\begin{array}{lll}
\mathsf{I} & ::= & \textbf{if}\ \mathsf{E}\ \textbf{then}\ \mathsf{I}\ \textbf{else}\ \mathsf{I} \\
\mathsf{I} & ::= & \textbf{if}\ \mathsf{E}\ \textbf{then}\ \mathsf{I} \\
\mathsf{I} & ::= & \textbf{skip}
\end{array}
$$

- Montrer que cette grammaire est ambiguë.
- Quelle est la convention usuellement adoptée ?
- Modifier la grammaire pour la rendre non ambiguë.
- Comment indiquer des précédences sur la grammaire intiale pour lever l'ambiguïté ?

Solution

Exercice

$$
\begin{array}{lll}
E & ::= & E\ E \\
E & ::= & E + E \\
E & ::= & \mathbf{id}
\end{array}
$$

- Comment déclarer les précédences pour que l'application et l'addition associent à gauche et que l'application ait une précédence plus forte que l'addition.

Solution

Sommaire

1. Analyse syntaxique
   - Introduction
   - Analyse descendante
   - Analyse ascendante
   - Conflits et Précédences
   - Arbres de syntaxe abstraite
   - Conclusion

Arbre de syntaxe abstraite

- Les constructions essentielles du langage
  - pas de construction pour les parenthèses
- Factoriser les constructions pour diminuer le nombre de cas à examiner
- reconnaitre et éliminer le “sucre syntaxique”
  ex: boucles for/while en C ou Java
- Localiser les erreurs

Localisation des erreurs

```
type location = Lexing.position * Lexing.position
type binop = Sum | Diff | Prod | Quot
type expr = { e_desc : loc_expr; e_loc : location}
and loc_expr =
    Cst of int
  | Var of string
  | Op of binop*expr*expr
  | Letin of string*expr*expr
type instr = { i_desc : loc_instr; i_loc : location}
and loc_instr = Set of string*expr | Print of expr
type prg = instr list
```

Vers l’analyse sémantique

L’arbre de syntaxe abstraite issu de l’analyse syntaxique correspond à l’entrée ”brute” de l’utilisateur:

L’analyse sémantique va servir à glaner des informations:

- Relier les déclarations d’identifiant et leur utilisation.
  On utilisera alors des identifiants uniques ou bien des entiers pour plus d’efficacité.
- Calculer des informations de type sur chaque sous-expression.
- Distinguer certaines constructions (quand il y avait surcharge syntaxique) ou au contraire en éliminer (si redondance).
- ...

Implantation des arbres

Remarque:

- Les différences entre les arbres de syntaxe abstraite (ast) après analyse syntaxique et les ast après décoration par l'analyse sémantique peuvent justifier d'avoir deux types d'ast distincts (même s'ils partagent de nombreuses constructions).
- Si les deux structures sont suffisament similaires, on peut utiliser un même type ocaml éventuellement polymorphe.

Sommaire

1. Analyse syntaxique
   - Introduction
   - Analyse descendante
   - Analyse ascendante
   - Conflits et Précédences
   - Arbres de syntaxe abstraite
   - Conclusion

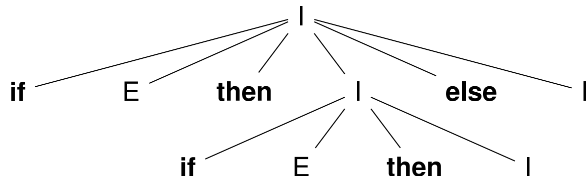
Conclusion

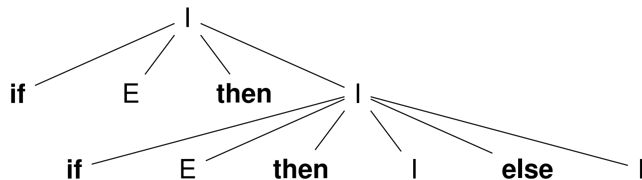
Déterminer ce qui peut/doit être réalisé à chaque étape:

- Analyse lexicale : identifier ce qui joue un rôle dans la grammaire, supprimer les espaces, commentaires.
- Analyse syntaxique : la structure générale du programme.
- Analyse sémantique : les analyses *fines*.

Construction if-then-else

Ambiguité sur la phrase: **if** *E* **then if** *E* **then** *I* **else** *I*





Résolution du conflit

Grammaire non ambiguë

- expressions *IT* qui ont (au moins) un **then** qui n'est pas associé à un **else**
- expressions *ITE* dont chaque **then** est associé à un **else**;
- seule une expression *ITE* peut aller entre **then** et un **else**.

$$
\begin{array}{lcl}
I & ::= & IT \\
I & ::= & ITE \\
IT & ::= & \textbf{if } E \textbf{ then } I \\
IT & ::= & \textbf{if } E \textbf{ then } ITE \textbf{ else } IT \\
ITE & ::= & \textbf{if } E \textbf{ then } ITE \textbf{ else } ITE \\
ITE & ::= & \textbf{skip}
\end{array}
$$

Résolution du conflit

Précédences:

```
    10: shift/reduce conflict (shift 11, reduce 2) on ELSE
state 10
        instr : IF expr THEN instr . ELSE instr  (1)
        instr : IF expr THEN instr .  (2)

        ELSE  shift 11
        $end  reduce 2
```

Il faut choisir *shift* donc donner une précédence plus forte à **else** qu'à **then**.

Retour

précédence de l'application

```
%left PLUS
%nonassoc IDENT
%nonassoc app
%%
expr:
| expr expr %prec app { }
| expr PLUS expr      { }
| IDENT               { }
;
```

Cas de conflits possibles

```
expr expr. PLUS        - reduce
expr expr. IDENT       - reduce
expr PLUS expr. PLUS   - reduce
expr PLUS expr. IDENT  - shift
```

La précédence de `IDENT` doit être supérieure à celle de `PLUS` et toutes deux inférieures à celle de la règle d'application.

Retour