Cours de Compilation-Exercices

Master d'Informatique M1 2011–2012

3 octobre 2011

3 Analyse syntaxique

3.1 Analyse ascendante

(D'après examen septembre 2000)

On se donne un langage de types permettant de décrire le type des entiers ou celui de fonctions à valeur entière, prenant en argument des entiers ou d'autres fonctions de même nature.
Un type est donc soit la constante `int` soit de la forme $\tau_1 * \ldots * \tau_n \rightarrow$ `int` avec τ_i des types.
Pour reconnaître ce langage de types, on se donne la grammaire suivante avec comme ensemble de terminaux $\{\sharp, \texttt{->}, *, \texttt{int}\}$ et comme ensemble de non-terminaux $\{S, A, T\}$ avec S le symbole de départ:

S	::=	T $\sharp$
T	::=	`int`
T	::=	A `->` `int`
A	::=	T
A	::=	T * A

1. Donner les étapes de l'analyse ascendante du mot `int -> int * int -> int` $\sharp$.
   Construire l'arbre de dérivation syntaxique correspondant.
2. Quels terminaux peuvent suivre les symboles non-terminaux T et A dans une dérivation.
3. Les états de l'analyse SLR(1) de cette grammaire sont les suivants:

s_1:
- $S \rightarrow .T\sharp$
- $T \rightarrow .$`int`
- $T \rightarrow .A$`->int`
- $A \rightarrow .T$
- $A \rightarrow .T * A$

s_2:
- $T \rightarrow$ `int.`

s_3:
- $S \rightarrow T.\sharp$
- $A \rightarrow T.$
- $A \rightarrow T. * A$

s_4:
- $T \rightarrow A.$`->int`

s_5:
- $A \rightarrow T * .A$
- $T \rightarrow .$`int`
- $T \rightarrow .A$`->int`
- $A \rightarrow .T$
- $A \rightarrow .T * A$

s_6:
- $T \rightarrow A$`->.int`

s_7:
- $A \rightarrow T.$
- $A \rightarrow T. * A$

s_8:
- $A \rightarrow T * A.$
- $T \rightarrow A.$`->int`

s_9:
- $T \rightarrow A$`->int.`

   – Construire la table de transitions. Pour chaque symbole x terminal ou non, un état qui contient un ensemble d'items $X \rightarrow m_1.xm_2$ évolue vers un état qui contient tous les items $X \rightarrow m_1x.m_2$ par une transition étiquetée par x.
   On notera que si m_2 commence par un symbole non terminal Y, l'état d'arrivée contient également les items $Y \rightarrow .m$ pour chaque règle de production $Y ::= m$.
   – Indiquer les états présentant des conflits et dans ces états, les différentes actions possibles.
4. Expliquer la nature du conflit obtenu en donnant un exemple d'entrée où ce conflit se produit. La grammaire donnée est-elle ambiguë ?
5. Que suggérez-vous pour remédier à ce problème ?

6. En caml, la grammaire des types inclut les règles

S	:= T ♯
T	:= `int`
T	:= T `->` T
T	:= T `*` T

La précédence de `*` est plus importante que celle de `->`, le symbole `->` associe à droite.

(a) Donner l'arbre de dérivation pour l'expression: `int -> int * int -> int` ♯

(b) indiquer les déclarations de précédence à ajouter à la grammaire pour éliminer les conflits.

3.2 Conflits et précédences

(D'après partiel 2009) Certaines notations facilitent l'écriture de la grammaire des langages de programmation. Soit une grammaire G, `t` un terminal et M un mot formé de terminaux et de non terminaux :

- $\langle M \rangle_{\texttt{t}}^{+}$ représente le langage des mots de la forme $m_1 \texttt{t} \ldots \texttt{t} m_n$ tels que m_i est dérivable à partir de M.
- $\langle M \rangle_{\texttt{t}}^{*}$ représente le même langage que $\langle M \rangle_{\texttt{t}}^{+}$ auquel on ajoute le mot vide.

1. Compléter la grammaire G avec deux nouveaux non terminaux: M_0 à partir duquel on pourra dériver le langage $\langle M \rangle_{\texttt{t}}^{*}$ et M_1 à partir duquel on dérive le langage $\langle M \rangle_{\texttt{t}}^{+}$.
2. On utilise cette notation pour décrire un langage d'expressions fonctionnelles avec des déclarations locales multiples. Les terminaux de cette grammaire sont formés des mots-clés **fun**, **let**, **in**, **and**, des symboles `(`, `)`, `+`, `=` et $\rightarrow$, et des classes `id` et `cte` pour représenter les identificateurs et les constantes.

   expr := `id` | `cte` | `(`*expr*`)` | *expr*`+`*expr* | **fun** `id` $\rightarrow$ *expr* | *expr expr* | **let** $\langle$`id=`*expr*$\rangle_{\textbf{and}}^{+}$ **in** *expr*

   (a) Ecrire une grammaire `ocamlyacc` pour ce langage.

   (b) Cette grammaire a plusieurs sources d'ambiguité: nommez-en au moins deux.
3. On se donne la grammaire `ocamlyacc` suivante:

```
%token id op pg pd fun arr
%%
E : pg E pd      {}
  | E op E       {}
  | fun id arr E {}
  | E E          {}
  | id           {}
;
```

`ocamlyacc` indique des conflits shift/reduce dans trois états. Le fichier d'information contient les éléments suivants sur la grammaire :

```
1  E : pg E pd
2    | E op E
3    | fun id arr E
4    | E E
5    | id
```

et sur un des états de conflit:

```
14: shift/reduce conflict (shift 3, reduce 3) on id
14: shift/reduce conflict (shift 9, reduce 3) on op
14: shift/reduce conflict (shift 4, reduce 3) on pg
14: shift/reduce conflict (shift 5, reduce 3) on fun
state 14
E : E . op E  (2)
E : fun id arr E .  (3)
E : E . E  (4)
```

(a) Donner un exemple d'entrée pour chacun des conflits de l'état précédent.

(b) Donner pour l'exemple correspondant au premier conflit sur `id` les dérivations dans le cas où on choisit la lecture et dans le cas où on choisit la réduction.

(c) Par défaut, que fait l'analyseur engendré par `ocamlyacc` ? Dans un conflit entre réduction et lecture, `ocamlyacc` choisit la lecture.

(d) On souhaite que l'application $E\,E$ ait une précédence plus forte que les opérations binaires $E\,\texttt{op}\,E$ qui auront elles-mêmes une précédence plus forte que la construction de fonction **fun** $id \to E$. On demande également que les opérations binaires et l'application associent à gauche. Ainsi le programme **fun** $f \to f\,1\,3+2$ sera compris comme **fun** $f \to (((f\,1)\,3)+2)$.

Comment modifier la grammaire `ocamlyacc` avec des précédences afin de supprimer les conflits en respectant les règles de précédence proposées.

3.3 Conflits

On se donne un langage pour reconnaître des expressions de type de tableaux soit effectives comme `int[4]` soit virtuelles comme `int[]` ou `void[]`.

Les terminaux de la grammaire sont {`int`,`char`,`void`,`[`,`]`,`number`}

Les règles de la grammaire sont :

S	:=	TE | TV
TE	:=	TET TEA
TET	:=	`int` | `char`
TEA	:=	`[` `number` `]`
TV	:=	TVT TVA
TVT	:=	`int` | `char` | `void`
TVA	:=	`[` `]`

Si cette grammaire est entrée dans yacc, on obtient le message suivant:

```
2 rules never reduced
2 reduce/reduce conflicts.
```

- Expliquer pourquoi (on pourra s'aider de l'automate engendré par Ocamlyacc).
- Proposer une grammaire équivalente dans lequel ce problème est résolu.

3.4 Analyse syntaxique, conflits

(Examen sesion 2 - juin 2011)

On considère un langage avec trois symboles terminaux `CTE`, `ARRAY` et `TIMES`, `LP`, `RP` et les règles suivantes :

```
typ :
| CTE                                { }
| typ ARRAY                          { }
| typ TIMES typ                      { }
| LP typ RP                            { }
;
```

Ocamlyacc indique deux conflits shift/reduce dans l'état suivant

```
10: shift/reduce conflict (shift 7, reduce 3) on ARRAY
10: shift/reduce conflict (shift 8, reduce 3) on TIMES
state 10
typ : typ . ARRAY  (2)
typ : typ . TIMES typ  (3)
typ : typ TIMES typ .  (3)

ARRAY  shift 7
TIMES  shift 8
$end  reduce 3
RP  reduce 3
```

1. Donner pour chacun des conflits un exemple d'entrée sur laquelle le conflit se produit.
2. On souhaite que l'opérateur `TIMES` associe à gauche et que le symbole `ARRAY` ait une précédence plus forte que `TIMES`. Indiquer dans l'état précédent, quelle action (shift ou reduce) doit être faite lorsque le symbole à lire est `ARRAY` et lorsque c'est `TIMES` pour respecter ces conventions.
3. Compléter la grammaire avec le choix de précédence sur les terminaux pour supprimer les conflits dans la grammaire.