Université de Nice-Sophia Antipolis — Licence d'Informatique
Module d'analyses — 2004–2005

Examen de septembre 2005

**Durée :** 3 heures
Tous documents autorisés

# 1 Énoncé conditionnel

Grammaire $G_0$

Axiome = $S$

$N_0 = \{S, C\}$

$T_0 = \{a, b, \texttt{if}, \texttt{then}, \texttt{else}\}$

$P_0 = \{\ S \to a$
$S \to \texttt{if}\ b\ \texttt{then}\ SC$
$C \to \nu$
$C \to \texttt{else}\ S$ $\}$

**a.** Construisez l'automate LR(0) reconnaissant $G_0$. Ajoutez les contextes LALR(1) sur cet automate.

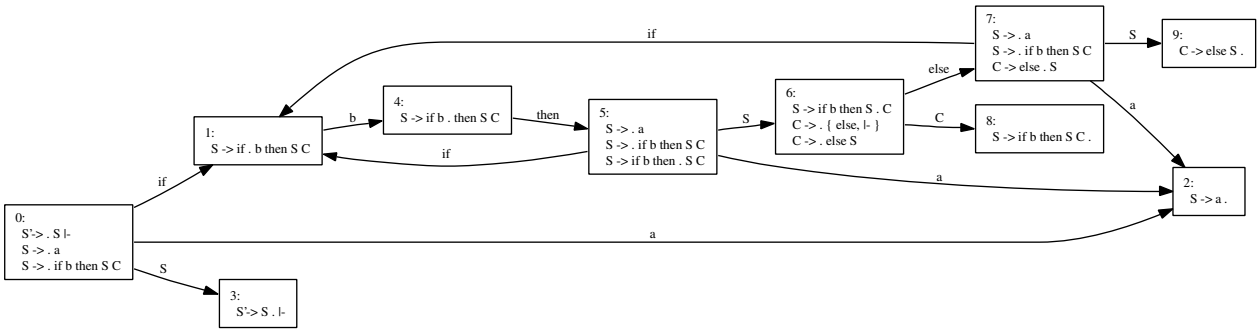


FIG. 1 – Automate LALR(1) reconnaissant $G_0$.

Le seul état inconsistant de l'automate LR(0) est l'état 6, où l'on a le choix entre le décalage de `else` et la réduction de $\nu$ à $C$. L'ensemble des contextes pour cette réduction est donné par

$$(q_6, C \to\uparrow) \text{ suite } \{(q_6, C)\} \text{ inclus } (q_0, S) \text{ lit } \{\vdash\} \text{ et} \tag{1}$$
$$(q_6, C \to\uparrow) \text{ suite } \{(q_6, C)\} \text{ inclus } (q_5, S) \text{ lit } \{\texttt{else}\}. \tag{2}$$

. . . . . .

**b.** La grammaire $G_0$ est-elle

1. LR(0) ?
2. LALR(1) ?
3. SLR(1) ?
4. LL(1) ?

Justifiez à chaque fois votre réponse.

---

1. L'automate LR(0) de la figure 1 n'est pas déterministe ; $G_0$ n'est pas LR(0).
2. L'automate LALR(1) de la figure 1 n'est pas déterministe ; $G_0$ n'est pas LALR(1).
3. Comme $G_0$ n'est pas LALR(1), elle n'est pas non plus SLR(1).
4. On a pour les productions $C \to \nu$ et $C \to \texttt{else}\ S$

$$\text{Premier}_1(\nu\ \text{Suivant}_1(C)) = \{\texttt{else}, \vdash\} \text{ et} \tag{3}$$
$$\text{Premier}_1(\texttt{else}\ S\ \text{Suivant}_1(C)) = \{\texttt{else}\}\ ; \tag{4}$$

   $G_0$ n'est donc pas LL(1).

. . . . . .

**c.** Donnez les deux séquences de dérivations *droites* de l'axiome $S$ pour la chaîne terminale

$$\texttt{if}\ b\ \texttt{then}\ \texttt{if}\ b\ \texttt{then}\ a\ \texttt{else}\ a. \tag{5}$$

Qu'en déduisez-vous ? La grammaire $G_0$ est-elle LR(2) ?

---

$$\begin{aligned} S &\Rightarrow \texttt{if}\ b\ \texttt{then}\ SC \\ &\Rightarrow \texttt{if}\ b\ \texttt{then}\ S \\ &\Rightarrow \texttt{if}\ b\ \texttt{then}\ \texttt{if}\ b\ \texttt{then}\ SC \\ &\Rightarrow \texttt{if}\ b\ \texttt{then}\ \texttt{if}\ b\ \texttt{then}\ S\ \texttt{else}\ S \\ &\Rightarrow \texttt{if}\ b\ \texttt{then}\ \texttt{if}\ b\ \texttt{then}\ S\ \texttt{else}\ a \\ &\Rightarrow \texttt{if}\ b\ \texttt{then}\ \texttt{if}\ b\ \texttt{then}\ a\ \texttt{else}\ a \end{aligned} \tag{6}$$

$$\begin{aligned} S &\Rightarrow \texttt{if}\ b\ \texttt{then}\ SC \\ &\Rightarrow \texttt{if}\ b\ \texttt{then}\ S\ \texttt{else}\ S \\ &\Rightarrow \texttt{if}\ b\ \texttt{then}\ S\ \texttt{else}\ a \\ &\Rightarrow \texttt{if}\ b\ \texttt{then}\ \texttt{if}\ b\ \texttt{then}\ SC\ \texttt{else}\ a \\ &\Rightarrow \texttt{if}\ b\ \texttt{then}\ \texttt{if}\ b\ \texttt{then}\ S\ \texttt{else}\ a \\ &\Rightarrow \texttt{if}\ b\ \texttt{then}\ \texttt{if}\ b\ \texttt{then}\ a\ \texttt{else}\ a \end{aligned} \tag{7}$$

La grammaire $G_0$ est ambigüe. Elle n'est pas LR(2), ni LR($k$) pour aucune valeur de $k$.

. . . . . .

**d.** Quel est le comportement de `bison` avec la grammaire $G_0$ sur la chaîne terminale (5) ?

---

En présence du conflit entre décalage et réduction, `bison` choisira le décalage, interprétant la chaîne terminale (5) comme issue de la séquence de dérivations (6).

. . . . . .

**e.** Les énoncés conditionnels sont un problème d'analyse syntaxique courant. Proposez un changement simple de la grammaire et du langage qui permet d'éviter cet écueil (il est possible de ne pas changer le langage).

---

Il suffit d'ajouter un mot-clef marquant la fin de l'énoncé conditionnel.

Grammaire $G'_0$

Axiome = $S$

$N'_0 = \{S, C\}$

$T'_0 = \{a, b, \texttt{if}, \texttt{then}, \texttt{else}, \texttt{fi}\}$

$P'_0 = \{$
$S \to a$
$S \to \texttt{if}\ b\ \texttt{then}\ SC\ \texttt{fi}$
$C \to \nu$
$C \to \texttt{else}\ S$
$\}$

Des solutions utilisant des accolades pour délimiter les blocs ont le même effet.

On peut écrire une grammaire équivalente à $G_0$ et qui impose le choix de `bison`. La grammaire $G''_0$ est SLR(1).

Grammaire $G''_0$

Axiome = $S$

$N''_0 = \{S, C\}$

$T''_0 = \{a, b, \texttt{if}, \texttt{then}, \texttt{else}\}$

$P''_0 = \{$
$S \to a$
$S \to \texttt{if}\ b\ \texttt{then}\ S$
$S \to \texttt{if}\ b\ \texttt{then}\ C\ \texttt{else}\ S$
$C \to a$
$C \to \texttt{if}\ b\ \texttt{then}\ C\ \texttt{else}\ C$
$\}$

. . . . . .

## 2 *Uniform Resource Identifiers*

Une URI est une chaîne de caractères désignant une ressource. La forme des URI est standardisée par l'IETF dans la RFC 3986 `[http ://www.ietf.org/rfc/rfc3986.txt]`. Nous allons nous baser sur ce standard pour faire reconnaître une URI valide utilisant le protocole HTTP à `flex`.

Le découpage d'une URI se fait ainsi :

```
 http://exemple.fr:8042/par/ici?nom=furet#nez
 \__/   \_____________/\_____/ \_______/ \_/
   |          |          |        |      |
schéma     autorité    chemin   requête fragment
```

Voici la norme simplifiée que vous allez implémenter :

- Comme nous ne nous intéressons qu'au protocole HTTP, le *schéma* sera toujours `http`, potentiellement en majuscules et minuscules mêlées.
- Le *schéma* est séparé de l'*autorité* par la chaîne de caractères « `://` ».
- L'*autorité* est le nom de l'hôte (`exemple.fr` dans l'exemple), optionellement suivi par le caractère deux points « `:` » et un numéro de port, qui est une succession de chiffres. Le nom de l'hôte est une séquence potentiellement vide de caractères, les caractères deux points « `:` » et arobasse « `@` » exclus.
- Le *chemin* est une séquence potentiellement vide de segments précédés d'un slash « `/` » chacun. Chaque segment est une séquence potentiellement vide de caractères.
- La *requête* comme le *fragment* sont des séquences potentiellement vides de caractères, potentiellement mêlés avec des slashs « `/` » ou des points d'interrogation « `?` ». La *requête* comme le *fragment* sont falcutatifs : la *requête*, si elle existe, est précédée par un point d'interrogation « `?` » (qui ne fait pas partie du texte de la requête), et le *fragment*, s'il existe, est précédé par une dièse « `#` » (qui ne fait pas partie du texte du fragment).
- Les caractères considérés sont :
  - les lettres,

- les chiffres,
- les deux points « `:` », l'arobasse « `@` »,
- les délimiteurs « `~` », « `_` », « `.` », « `-` », « `!` », « `$` », « `&` », « `'` », « `(` », « `)` », « `*` », « `+` », « `,` », « `;` » et « `=` »,
- et les caractères encodés dénotés par leur code sur deux chiffres hexadécimaux précédé par le caractère pourcent « `%` ».

**f.** Écrivez le contenu d'un fichier `flex` permettant de vérifier si le texte lu forme une URI valide selon cette norme ou non.

---

```
/* uri.l
 * Reconnaissance d'une URI valide.
 * À compiler avec l'option '-i' de flex.
 */
%option noyywrap

pchar     {char}|[:@]
char      [[:alnum:]~_.\-!$&'()*+,;=]|{code}
code      %[[:xdigit:]][[:xdigit:]]

segment   \/{pchar}*
sequence  ({pchar}|[?/])*
hote      {char}*
port      :[[:digit:]]*

schema    http
autorite  {hote}{port}?
chemin    {segment}*
requete   \?{sequence}
fragment  #{sequence}

uri       {schema}"://"{autorite}{chemin}{requete}?{fragment}?

%%
{uri}     printf ("Valid URI: <%s>.\n", yytext);
.*        printf ("Not a valid URI.\n");
%%

int
main (void)
{
  return yylex ();
}
```

.....

# 3 Descente récursive

Voici un fragment de la grammaire (simplifiée) du langage Pascal :

```
énoncé = variable := expression
       | appel_de_procédure
       | 'tantque' expression 'faire' énoncé
```

```
       | 'répéter' énoncé [ ';' énoncé ] 'jusque' expression
       | 'début' énoncé [ ';' énoncé ] 'fin'
       | 'si' expression 'alors' énoncé { 'sinon' énoncé }
variable = 'ident' { références }
références = '[' expression [ ',' expression ] ']'
appel_de_procédure = variable { paramètres }
paramètres = '(' expression [ ',' expression ] ')'
```

Les parties entre crochets correspondent à une répétition de 0 à $n$ fois, les parties entre accolades correspondent à une partie facultative. Les terminaux sont entre apostrophes.

**g.** Construisez l'analyseur $LL(1)$ par descente récursive correspondant à cette grammaire. Efforcez-vous de rester au plus près de la grammaire donnée, si ce n'est que vous n'avez probablement pas besoin d'une procédure `références` ni d'une procédure `paramètres`.

En d'autres termes, vous ne devez probablement avoir que les procédures `énoncé`, `variable` et `appel_de_procédure`.

**h.** Avez-vous remarqué que cette grammaire n'est pas $LL(1)$ ? Montrez qu'elle n'est même pas $LL(k)$.

**i.** Parmi les solutions possibles pour résoudre le problème vu dans l'exercice précédent, les compilateurs pour le langage Pascal écrits par descente récursive privilégient d'ordinaire l'utilisation d'attributs, ou de l'équivalent. Proposez une solution en ce sens. Supposez, comme c'est le cas en Pascal, que tous les identificateurs sont déclarés avant d'être utilisés.