

# Partiel de compilation, 18 Nov. 2004

durée: 2h, documents autorisés

**Attention: version corrigée**

## 1 Procédure

Considérons le programme C de la Figure 1. On utilisera la structure d'un enregistrement d'activation (AR) présentée en figure 2 lors de l'appel d'une procédure. Par convention, le pointeur sur l'AR (que l'on note ARP) pointe sur le mot contenant l'adresse de l'AR de la procédure appelante. On rappelle que l' *ARP appelant* pointe sur l'enregistrement d'activation de la procédure appelante alors que le *lien d'accès* pointe sur la procédure englobante lexicalement.

```
1  int main() {
2      void A(int i) {
3          void B(int i) {
4              void C(int i) {
5                  /* code de C */
6                  A(i-1);
7              }; /* fin C */
8              /* code de B */
9              C(i);
10             }; /* fin B */
11             /* code de A */
12             if (i<=0) printf("OK\n");
13             else B(i);
14         }; /* fin A */
15
16         /* code de main */
17         A(1);
18     } /* fin main */
```

Figure 1: Programme C considéré

1. Dessiner l'état de la pile résultant de l'exécution du code au moment où le programme affiche OK sur la sortie standard (ligne 12). On dessinera la pile en indiquant en particulier où pointent: le pointeur sur l'enregistrement d'activation (ARP) et le lien d'accès.
2. Lorsque la procédure C appelle la procédure A, le lien d'accès écrit dans l'AR de la procédure A doit pointer sur l'AR de `main` qui est la procédure englobante lexicalement. Ce lien d'accès est mis en place à l'exécution, mais le code assembleur exécuté est, bien sûr, généré à la compilation.

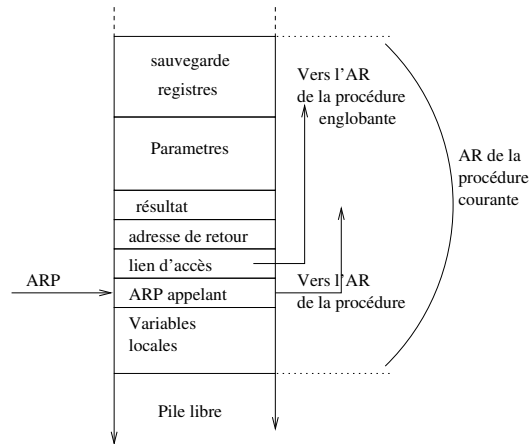


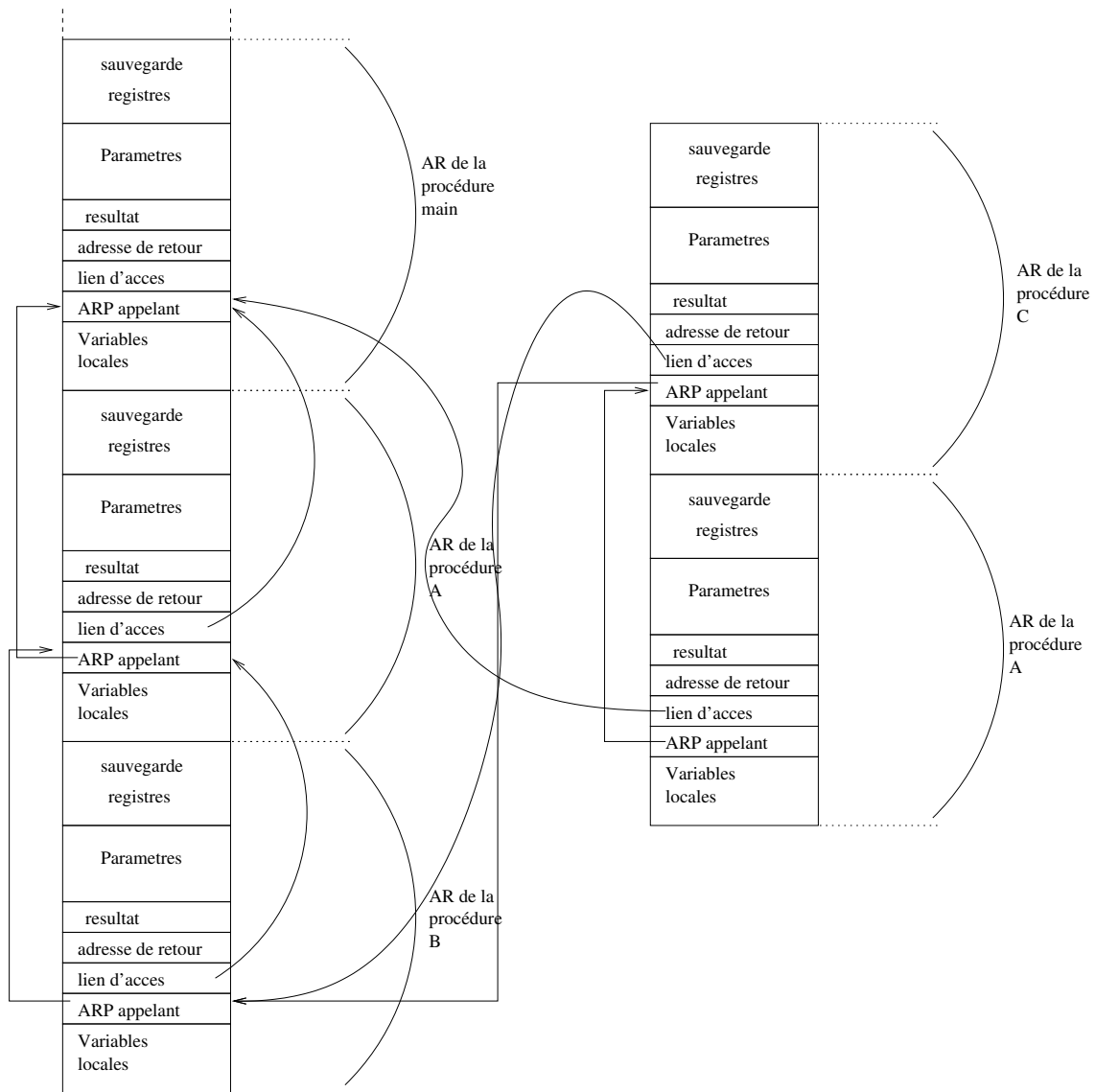
Figure 2: Structure d'un enregistrement d'activation d'une procédure

a) Est ce que le code assembleur qui effectue cette action à l'exécution peut se trouver dans le code de la procédure appelée (code de A)?

b) proposer un mécanisme général pour mettre en place le liens d'accès et donner sur cet exemple particulier (appel de A par C sur le code de la figure 1) le code qui pourrait être généré par la compilateur.

**Corrigé :**

1. Voici l'état de la pile lorsque A écrit OK: Main appelle A qui appelle B qui appelle C qui appelle A.



## 2. on dérecursive et on factorise a gauche

a) Non le code assembleur ne peut pas se trouver dans code de la procédure appelée **A** car on a alors perdu les informations du contexte dans lequel cette procédure est appelée. Pour retrouver l'AR de **main**, il faut savoir que l'on est dans une procédure de niveau 3 (**C**) et que l'on appelle une procédure de niveau 1 (**A**). Selon que **A** est appelé d'une procédure de niveau 1, 2 ou 3, le code a généré n'est pas le même.

b) Une méthode possible: dans la procédure **C** on remonte trois fois le long du lien d'accès (différence entre les niveaux de **A** et **C** plus 1) et on copie l'adresse sur la pile. Le code Iloc sera donc (dans la procédure **C**):

<i>loadAI</i> $r_{arp}, 4$	$\Rightarrow r_1$	<i>//</i> $r_1 \leftarrow ARP(B)$
<i>loadAI</i> $r_1, 4$	$\Rightarrow r_2$	<i>//</i> $r_2 \leftarrow ARP(A)$
<i>loadAI</i> $r_2, 4$	$\Rightarrow r_3$	<i>//</i> $r_3 \leftarrow ARP(main)$
<i>storeAI</i> $r_3$	$\Rightarrow r_{sp}, -4$	<i>//push</i> $ARP(main)$
<i>subI</i> $r_{sp}, 4$	$\Rightarrow r_{sp}$	

## 2 Grammaire

On considère la grammaire  $G = \{T, NT, S, P\}$  suivante:

$T = \{id, (, ), ;, eof\}$ ,  $NT = \{E, T, L\}$ ,  $S = E$ ,

$$P = \left\{ \begin{array}{l} E \rightarrow E + T \\ \quad | \quad T \\ T \rightarrow id \\ \quad | \quad id(L) \\ L \rightarrow E ; L \\ \quad | \quad E \end{array} \right\}$$

1. Pourquoi n'est-elle pas  $LL(1)$ ?
2. Donner une grammaire  $G'$  équivalente qui est  $LL(1)$ . On ne demande pas de prouver formellement que les grammaires sont équivalentes, en revanche il faudra montrer précisément que votre grammaire est  $LL(1)$  (en calculant les ensembles *First* et *Follow* en particulier).

### Corrigé :

1. La grammaire est récursive à gauche donc elle n'est pas  $LL(1)$ .
  - (a) dérécursivation de  $E$ : on ajoute un non-terminal  $E'$ .

$$P = \left\{ \begin{array}{l} E \rightarrow T E' \\ E' \rightarrow + T E' \\ \quad | \quad \epsilon \\ T \rightarrow id \\ \quad | \quad id(L) \\ L \rightarrow E ; L \\ \quad | \quad E \end{array} \right\}$$

(b) factorisation à gauche: pour  $E$  dans  $L$  et  $id$  dans  $T$  on ajoute un non-terminal  $L'$  et  $T'$

$$P = \left\{ \begin{array}{l} E \rightarrow T E' \\ E' \rightarrow + T E' \\ \quad | \quad \epsilon \\ T \rightarrow id T' \\ T' \rightarrow \epsilon \\ \quad | \quad ( L ) \\ L \rightarrow E L' \\ L' \rightarrow ; L \\ \quad | \quad \epsilon \end{array} \right\}$$

(c) ensembles *First*: l'algorithme fixe les ensembles suivants:

<i>Symbol</i>	$E$	$E'$	$T$	$T'$	$L$	$L'$
$First(Symbol)$	{ <i>id</i> }	{+, $\epsilon$ }	{ <i>id</i> }	{(, $\epsilon$ }	{ <i>id</i> }	{;, $\epsilon$ }

Comme certains ensembles comportent  $\epsilon$ , on est obligé de calculer les ensembles *Follow*.

<i>Symbol</i>	$E$	$E'$	$T$	$T'$	$L$	$L'$
$Follow(Symbol)$	{ <i>eof</i> , ;, )}	{ <i>eof</i> , ;, )}	{+, ;, <i>eof</i> }	{+, ;, <i>eof</i> }	{})}	{})}

On a  $First(T') \cap Follow(T') = First(E') \cap Follow(E') = First(L') \cap Follow(L') = \emptyset$  donc pour chacun de ses non terminaux, on peut choisir sans ambiguïté quelle règle utiliser si on connaît le symbole qui arrive.

### 3 Automate SLR

On s'intéresse maintenant à la grammaire suivante, où  $S'$  est le symbole de départ et  $op$  et  $x$  sont des terminaux:

$$\left\{ \begin{array}{l} S' \rightarrow S \\ S \rightarrow S op S \\ \quad | \quad x \end{array} \right.$$

1. construire l'automate  $LR(0)$  de la grammaire. On rappelle que les états de l'automate  $LR(0)$  sont constitués d'ensembles d'item de la forme:  $[A \rightarrow \alpha \bullet \beta]$ , c'est à dire que l'on ne tient pas compte du *lookahead symbol*. L'automate  $LR(0)$  a donc pour état initial l'état  $E_1 = closure(\{[S' \rightarrow \bullet S]\}) = \{[S' \rightarrow \bullet S], [S \rightarrow \bullet S op S], [S \rightarrow \bullet x]\}$
2. Pouvez-vous déduire de votre automate que cette grammaire  $G$  est  $LR(0)$ ?
3. Donner un argument pour montrer que cette grammaire n'est pas  $LR(1)$  non plus.

**Corrigé :**

1. construire l'automate

2. on tombe sur un état  $E = \{[S \rightarrow S \text{ op } S \bullet], [S \rightarrow S \bullet \text{ op } S]\}$  qui contient un conflit shift/reduce.
3. on peut construire deux arbres syntaxique pour  $x \text{ op } x \text{ op } x$

## 4 Expressions régulières

On considère l'expression régulière qui reconnaît les nombres binaires:

$$1(0|1)^* | 0$$

On a vu en cours que l'on pouvait construire de manière inductive un automate non-déterministe qui reconnaît une expression régulière donnée. Les règles constructives utilisées en cours sont données dans la colonne *règles initiales* de la Figure 3.

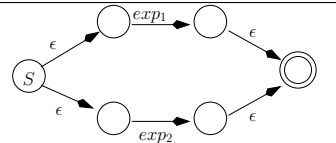
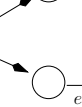
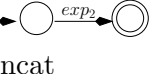
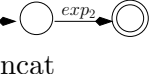




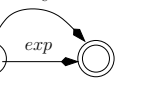
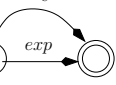


opérateurs	règles initiales	règles simplifiées
$exp_1 exp_2$	 <p>regle_ou</p>	 <p>regle_ou_simplifiée</p>
$exp_1.exp_2$	 <p>regle_concat</p>	 <p>regle_concat_simplifiée</p>
$exp^+$	 <p>regle_plus</p>	 <p>regle_plus_simplifiée</p>
$exp^*$	 <p>regle_etoile</p>	 <p>regle_etoile_simplifiée</p>
$exp?*$	 <p>regle_interrogation</p>	 <p>regle_interrogation_simplifiée</p>
$\forall a \in \Sigma$		

Figure 3: Règles constructives pour obtenir l'automate non-déterministe correspondant à une expression régulière

Ainsi à partir de notre expression représentant les nombres binaires, on obtient l'automate non-déterministe de la Figure 4.

- Déterminez l'automate.

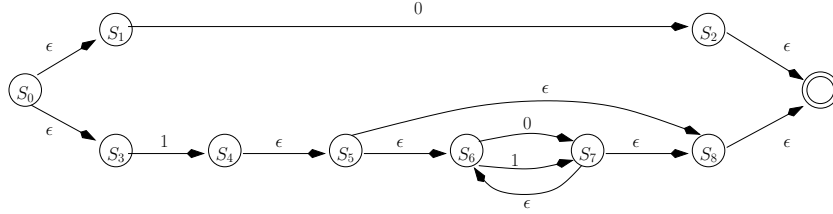
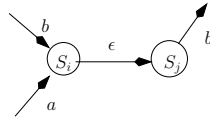


Figure 4: Automate non-deterministe reconnaissant l'expression  $1(0|1)^* | 0$ .

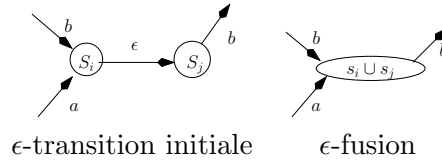
On appelle “ $\epsilon$ -transition”, une transition du type



Clairement, l'automate de la Figure 4 contient de nombreuses  $\epsilon$ -transitions inutiles. On se propose ici de réduire le nombre d' $\epsilon$ -transitions d'un automate ainsi obtenu à l'aide de deux différentes approches: dans la partie 4.1, on cherchera à éliminer au maximum *après coup* les  $\epsilon$ -transitions; dans la partie 4.2, on cherchera à construire des règles qui génèrent moins d' $\epsilon$ -transition.

#### 4.1 Réduction après coup

On appelle  $\epsilon$ -fusion la transformation sur l'automate qui fusionne deux états liés par une  $\epsilon$ -transition en fusionnant les arêtes sortantes et entrantes. C'est une fusion classique de graphe:



- Combien peut on enlever au maximum d' $\epsilon$ -transitions à l'automate de la Figure 4? Donner l'automate résultant: les noms des nouveaux états seront une union des noms des anciens états fusionnés.
- Trouvez une règle de réécriture locale (une règle qui tient compte de l'ensemble des successeurs et des prédécesseurs de chacun des deux états concernés, mais qui ne va pas plus loin) qui permette de fusionner 5 des  $\epsilon$ -transitions de l'automate de la Figure 4.

#### 4.2 Simplification des règles constructives

L'idéal serait d'utiliser à la place des règles initiales de la Figure 3, les *règles simplifiées* proposées dans cette même figure.

- Pourquoi la simplification correspondant aux règles *regle\_concat\_simplifiée* et *regle\_plus\_simplifiée* est incorrecte?
- Quelles sont les règles compatibles avec la règle *regle\_concat\_simplifiée*? Quelles sont celles compatibles avec la règle *regle\_plus\_simplifiée*

## 5 Bison

On considère un langage permettant de décrire des arbres binaires. Ces arbres ont un seul type de feuille : `ident` et deux types de noeuds internes `lambda` et `app`. les mots du langage suivent donc la grammaire suivante:

```
expr  →  app(expr,expr)
        |  lambda(expr,expr)
        |  ident
```

Soit maintenant les instructions Bison suivante:

```
%token ID LAMBDA
%type <Expression> Expr
```

```
%%
```

```
Expr : ID
      { $$=ident; }
    | '(' LAMBDA ID '.' Expr ')'
      { $$ = lambda( ident, $5 ); }
    | '(' Expr Expr ')'
      { $$ = app( $2 , $3 ); }
```

Dessiner l'arbre de syntaxe abstraite résultant du parsing de l'expression suivante:

```
( ( LAMBDA ID . ( ID ID ) ) ( LAMBDA ID . ID ) )
```

**Corrigé**

```
app( lambda( ident , app( ident , ident ) ) , lambda( ident , ident ) )
```