Contrôle continu - Compilation (L3 Info)

Durée : 2h - documents autorisés

Aix-Montperrin, le 4 mars 2014

Conventions

— Axiome : symbole non terminal à gauche de la première production de la grammaire
— Symboles non terminaux : lettres *MAJUSCULES ITALIQUES*
— Symboles terminaux : lettres `minuscules true-type` ou caractères spéciaux simples

1 Problème : déclarations de variables de types complexes

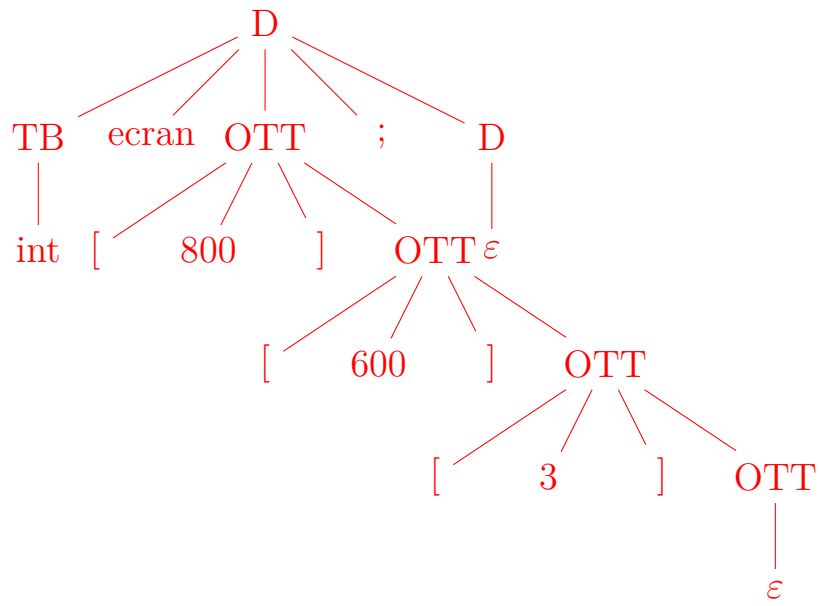
Pour les prochaines questions, on considérera la grammaire G_{dec} ci-dessous. En C, on peut déclarer des tableaux à plusieurs dimensions, c'est-à-dire des suites de cases d'un *type base*. Nous considérerons ici seulement les types base caractères (`char`, 1 octet), entiers (`int`, 4 octets) et réels (`double`, 8 octets). Par exemple, la déclaration `char phrase[ 20 ];` indique un tableau de caractères à une dimension, avec 20 cases de type `char`. La déclaration `int ecran[800][600][3];` indique un tableau d'entiers à trois dimensions, par exemple un écran de 800 lignes par 600 colonnes, avec 3 valeurs par pixel pour encoder les couleurs en RGB. La grammaire ci-dessous permet de déclarer des variables de type base ainsi que des tableaux composés de cases d'un type base.

(1) $D \rightarrow$ *TB* `idv` *OTT* `;` *D*
(2) $\quad\quad\;\; | \; \varepsilon$
(3) *TB* $\rightarrow$ `char`
(4) $\quad\quad\;\; |$ `int`
(5) $\quad\quad\;\; |$ `double`
(6) *OTT* $\rightarrow$ `[ nb ]` *OTT*
(7) $\quad\quad\;\; | \; \varepsilon$

L'axiome est une liste de déclarations D composées d'un type base TB, un identificateur de variable `idv`, une taille de tableau optionnelle OTT, point-virgule `;` et une autre déclaration D, possiblement vide. Un type base TB est un terminal `char`, `int` ou `double`. Une taille de tableau optionnelle OTT est une liste de nombres `nb` entre crochets `[` et `]`. La liste peut être vide, ce qui permet à la fois de déclarer des variables de type base et d'arrêter la liste des dimensions lors de l'application de la production récursive pour OTT.

Question 1 (5 pt) - Traduction dirigée par la syntaxe

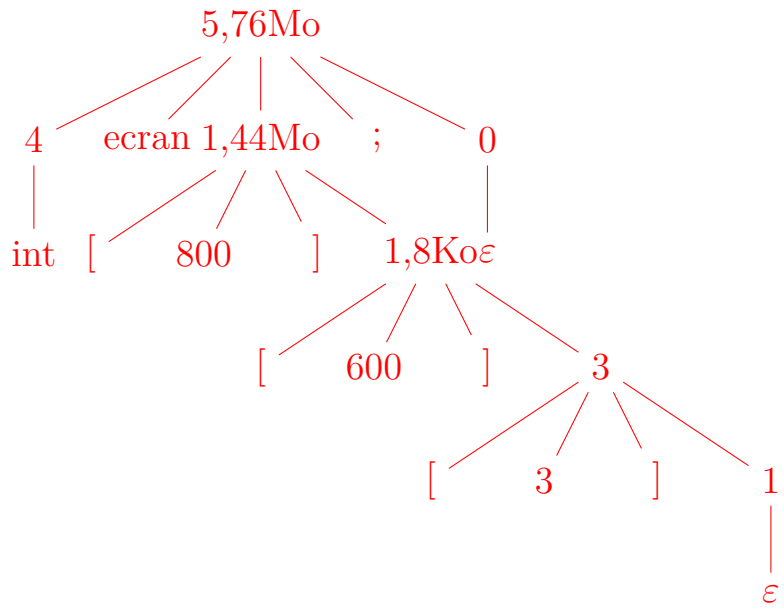
1. Dessinez l'arbre de dérivation de la déclaration `int ecran[ 800 ][ 600 ][ 3 ];` (1 pt)



2. Écrivez un schéma de traduction dirigée par la syntaxe qui associe à chaque production une action sémantique. Cette action a pour but de remplir l'attribut *no* (*nombre d'octets*) avec le nombre d'octets à allouer en mémoire pour un tableau multi-dimensionnel. (2 pt)

	Règle			Action sémantique
(1)	D	$\rightarrow$	TB `idv` OTT ; D_1	D.no = TB.no * OTT.to + D_1.no
(2)		|	ε	D.no = 0
(3)	TB	$\rightarrow$	`char`	TB.no = 1
(4)		|	`int`	TB.no = 4
(5)		|	`double`	TB.no = 8
(6)	OTT	$\rightarrow$	`[ nb ]` OTT_1	OTT.no = OTT_1.no * nb
(7)		|	ε	OTT.no = 1

3. Décorez l'arbre syntaxique de la question 1 avec les valeurs de l'attribut *no*. Combien vaut-il au nœud D, racine de l'arbre ? (1 pt)



4. Dans votre schéma, l'attribut *no* est-il synthétisé ou hérité ? Pourquoi ? (1 pt)
   Synthétisé parce que sa valeur dans un nœud dépend de la valeur aux nœuds fils.

Question 2 (6 pt) - Analyse $LL(1)$

1. Quels sont les symboles terminaux et non terminaux de G_{dec} ? (1 pt)
   $\Sigma = \{$ `char int double idv nb [ ] ;` $\}$
   $N = \{ TB\ OTT\ D \}$

2. Calculez les ensembles Premier pour les symboles non terminaux de la grammaire G_{dec}. (2 pt)
   Premier(D) = { `char int double` ε}
   Premier(TB) = { `char int double` }
   Premier(OTT) = { `[` ε}

3. Calculez les ensembles Suivant pour les symboles non terminaux de la grammaire G_{dec}. (2 pt)
   Suivant(D) = $\{\perp\}$
   Suivant(TB) = { `idv` }
   Suivant(OTT) = { `;` }

4. Construisez la table d'analyse $LL(1)$ de la grammaire G_{dec}. (1 pt)

	`char`	`int`	`double`	`[`	`]`	`;`	`idv`	$\perp$
D	1	1	1					2
TB	3	4	5					
OTT				6		7		

Question 3 (3 pt) - Écriture de grammaire (pointeurs et structures)

Nous voulons maintenant enrichir la grammaire G_{dec} avec les types *pointeur* et *structure*. Un *pointeur* est déclaré simplement en rajoutant une étoile `*` entre le nom du type et l'identificateur de la variable. Nous ne considérerons ici que les pointeurs de types base.

Pour les *structures*, on utilisera le mot-clé `struct`. Les structures sont des types complexes composés de cases nommées, et qui, contrairement aux tableaux, n'ont pas forcément toutes le même type. Par exemple, la déclaration `struct pixel{ int x; int y; char rgb[ 3 ]; }` représente un pixel de coordonnées entières `x` et `y`, avec un tableau de trois octets pour les composantes couleur `r`, `g` et `b`. On pourrait aussi définir un cercle de façon récursive, avec des structures imbriqués

```
struct cercle{
  struct centre{ int x; int y; };
  struct couleur{ char r; char g; char b; };
  double rayon;
};
```

1. Écrivez une grammaire hors contexte, extension de G_{dec}, qui permet de déclarer des variables de type base `char`, `int` et `double`, des tableaux et des pointeurs vers ces types base, ainsi que des structures (`struct`) contenant ces derniers. Les tableaux et les pointeurs sont construits à partir des types base. Les registres structures contenir des types base, des tableaux, des pointeurs et d'autres structures.

$$
\begin{array}{lcl}
D & \rightarrow & TB \ \texttt{idv} \ OTT \ \texttt{;} \ D \mid \\
 & & TB \ \texttt{*} \ \texttt{idv} \ \texttt{;} \ D \mid \\
 & & \texttt{struct} \ \texttt{idv} \ \texttt{\{} \ D \ \texttt{\}} \ \texttt{;} \ D \mid \varepsilon \\
TB & \rightarrow & \texttt{char} \mid \texttt{int} \mid \texttt{double} \\
OTT & \rightarrow & \texttt{[} \ \texttt{nb} \ \texttt{]} \ OTT \mid \varepsilon
\end{array}
$$

2 Questions théoriques

Question 4 (4 pt) - Récursivité gauche et ambiguïté

$$
G_1 : \begin{array}{lcl} P & \rightarrow & P \ \texttt{a} \mid I \ \texttt{b} \mid \texttt{b} \\ I & \rightarrow & I \ \texttt{a} \mid P \ \texttt{b} \mid \varepsilon \end{array}
$$

1. Décrivez le langage $L(G_1)$ engendré par la grammaire G_1. (1 pt)
   Les mots contenant des a et des b avec un nombre impair de b.
   $L = \{m \in \{a, b\}^* \mid |m|_b \mod 2 = 1\}$

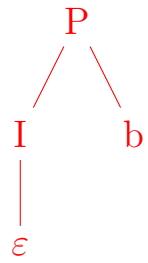
2. La grammaire G_1 est récursive à gauche. Écrivez une grammaire G_1' sans récursivité gauche directe ni indirecte, équivalente à G_1, c'est-à-dire telle que $L(G_1) = L(G_1')$. (2 pt)

$$G_1' : \begin{array}{lcl} P & \rightarrow & I \ \texttt{b} \ P' \mid \texttt{b} \ P' \\ P' & \rightarrow & \texttt{a} \ P' \mid \varepsilon \\ I & \rightarrow & \texttt{b} \ P' \ \texttt{b} \ I' \mid I' \\ I' & \rightarrow & \texttt{a} \ I' \mid \texttt{b} \ P' \ \texttt{b} \ I' \mid \varepsilon \end{array}$$

Les deux dernières productions peuvent être simplifiées par :

$$I \rightarrow \texttt{b} \ P' \ \texttt{b} \ I \mid \texttt{a} \ I \mid \varepsilon$$

3. La grammaire G_1 est-elle ambiguë ? Pourquoi ? (1 pt)
   Oui. Le mot `b` permet 2 arbres de dérivation.



Question 5 (2 pt) - Répondez aux questions de manière succincte (∼ 5 lignes)

1. Énumérez les principales différences entre un analyseur lexical et un analyseur syntaxique dans un compilateur. Pourrait-on construire un compilateur sans analyseur lexical ? (1 pt)
   L'analyseur lexical est simple, permet de mettre ensemble les caractères avec des automates finis ou des grammaires régulières. L'analyseur syntaxique est plus complexe, traite l'ordre des éléments et est généralement décrit par une grammaire hors contexte ou un automate à pile. On peut se passer d'analyseur lexical, mais le prix à payer est un analyseur syntaxique beaucoup plus complexe.

2. Pourquoi a t-on besoin de l'ensemble SUIVANT dans la construction de la table d'analyse $LL(1)$? (1 pt)
   Parce que quand ε est dans PREMIER d'un symbole, on doit regarder les prochains symboles qui suivent ce non-terminal pour appliquer la production vide.