UNIVERSITÉ BORDEAUX I — U.F.R. Mathématiques et Informatique
Session de Janvier 1996 — J.Bétréma

Maîtrise d'Informatique
Compilation (tronc commun)
(Durée: 2 heures)

Notes de cours et de TD autorisées. Autres documents interdits.

1 Analyse lexicale

Pour cet exercice, on définit un *mot* comme une suite de une ou plusieurs lettres minuscules, à l'exclusion de tout autre caractère. On partitionne les mots en deux ensembles :

- T est l'ensemble des mots qui se terminent par "tant" ; par exemple *autant* et *constatant* font partie de T ;
- U dénote l'ensemble complémentaire de T.

1. Ecrire un analyseur lexical, dans le format reconnu par *lex*, qui, dans un texte, affiche uniquement les mots de U (un par ligne).

   N.B. Comme la réponse est très courte, elle devra être rigoureusement exacte.

2. Construire, par la méthode de votre choix (que vous expliquerez), l'automate déterministe minimal qui reconnaît U.

   N.B. Comme, dans cette question, l'alphabet comporte 26 lettres, on utilisera une étiquette "autres cas", ou toute autre convention claire et compacte.

2 Analyse syntaxique

Soit G la grammaire des expressions suivante :

$$E \to E + T \mid T$$

$$T \to T * F \mid F$$

$$F \to (E) \mid \mathtt{id}$$

1. Un(e) étudiant(e) a la malencontreuse idée de simplifier G en remplaçant la production $F \to (E)$ par $T \to (E)$, après quoi il(elle) remplace F par `id` partout où F apparaît, et obtient ainsi la grammaire G_1 :

   $$E \to E + T \mid T$$
   $$T \to T * \mathtt{id} \mid (E) \mid \mathtt{id}$$

Donner un exemple simple d'expression reconnue par G et non reconnue par G_1, puis caractériser le langage reconnu par G_1.

Un analyseur ascendant pour la grammaire G_1 comporte-t-il moins de conflits que pour G ?

N.B. Sur la copie, décrire seulement les différences entre l'automate associé à G_1 et celui associé à G, sans perdre de temps à refaire les calculs, supposés connus, pour G. Cette recommandation vaut pour les questions suivantes.

2. On ajoute à G la production $F \to *F$, ce qui donne la grammaire G_2 :

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to *F \mid (E) \mid \mathtt{id}$$

G_2 est-elle ambiguë ? Donner d'abord une réponse de bon sens, tirée de votre expérience, puis prouver que G_2 est SLR(1).

3. Dans G_2, on remplace la production $F \to *F$ par $F \to F*$, ce qui fournit la grammaire G_3 ; G_3 est-elle ambiguë (on ne demande pas de preuve formelle) ? Prouver que G_3 n'est pas SLR(1). Donner un exemple simple d'expressions reconnues par G_3, et illustrant l'impossibilité d'une analyse ascendante LR(1).

4. Soit G_4 la grammaire (ambiguë) qui reconnaît le même langage que G_2 :

$$E \to E + E \mid E * E \mid \; *E \mid (E) \mid \mathtt{id}$$

On suppose connus les conflits, et leur traitement, pour l'analyseur ascendant déduit de la grammaire (ambiguë) équivalente à G ; quels sont les nouveaux conflits introduits par G_4, et comment les traiter ?

3 Attributs

Soit G la grammaire des expressions suivante :

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to \; (E) \mid \mathtt{id} \mid \mathtt{num}$$

On suppose que les identificateurs sont réduits aux lettres de l'alphabet latin, et que les constantes numériques représentent des entiers. On souhaite réaliser un traducteur qui effectue, autant que possible, les calculs sur ces constantes dans les expressions ; par exemple cet analyseur traduit $2 * 3 * x$ en $6 * x$.

Pour les calculs d'attributs demandés, employer un formalisme aussi précis que possible, mais on ne demande pas (vue la durée limitée de l'examen) le respect rigoureux du format reconnu par yacc, ni de la syntaxe C. On supposera que les fonctions élémentaires de *construction* d'arbres (ou de listes) sont disponibles : on ne demande aucun détail de réalisation de ces fonctions. Enfin les constructions demandées n'utilisent aucun algorithme de *parcours* d'arbres (ou de listes) ; en particulier, on ne demande pas l'affichage du résultat.

1. On commence par modifier le calcul de l'arbre syntaxique en effectuant, dans l'arbre, les opérations arithmétiques dès qu'on le peut. D'autre part, on tient compte du fait que 0 est élément neutre pour l'addition, et 1 pour la multiplication.

   Définir les attributs nécessaires, et leurs règles de calcul, pour construire cet arbre syntaxique simplifié.

   Donner des exemples simples où cette approche ne produit pas un arbre simplifié.

2. On change la représentation de l'arbre syntaxique, en tenant compte de l'associativité de l'addition et de la multiplication : un noeud interne de l'arbre contient un opérateur (addition ou multiplication), et pointe sur une liste de n opérandes ($n \geq 2$).

   Définir les attributs nécessaires, et leurs règles de calcul, pour produire un arbre où chaque liste d'opérandes comporte au plus une constante numérique (les opérateurs sont commutatifs).

   Donner des exemples simples où cette approche est plus efficace que celle de la question 1, et d'autres où elle n'effectue pas les simplifications souhaitables.

Corrigé

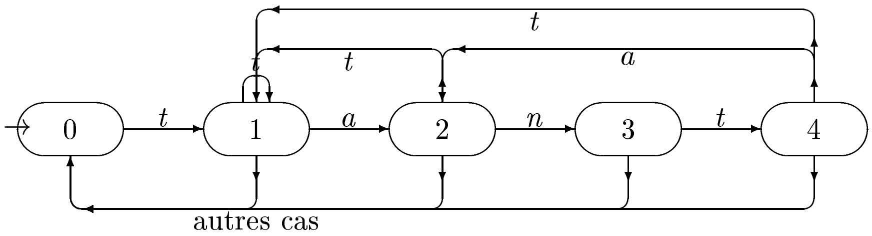
1 Analyse lexicale

1. Spécification en format *lex* :

```
%%
[a-z]*tant     ;
[a-z]+         printf ("%s\n", yytext);
.|\n           ;
%%
```

   Notes : un mot de T est reconnu par les deux premières règles, et *lex* donne la priorité à la première. Si un mot de U contient “tant” comme facteur (et non comme suffixe), c'est bien la seconde règle qui est exécutée, car *lex* reconnaît le plus long lexème.

2. Automate déterministe qui reconnaît T (on l'obtient par déterminisation, ou directement, en sachant que chaque état représente le plus long suffixe du mot lu, qui soit aussi un préfixe du motif “tant”) :

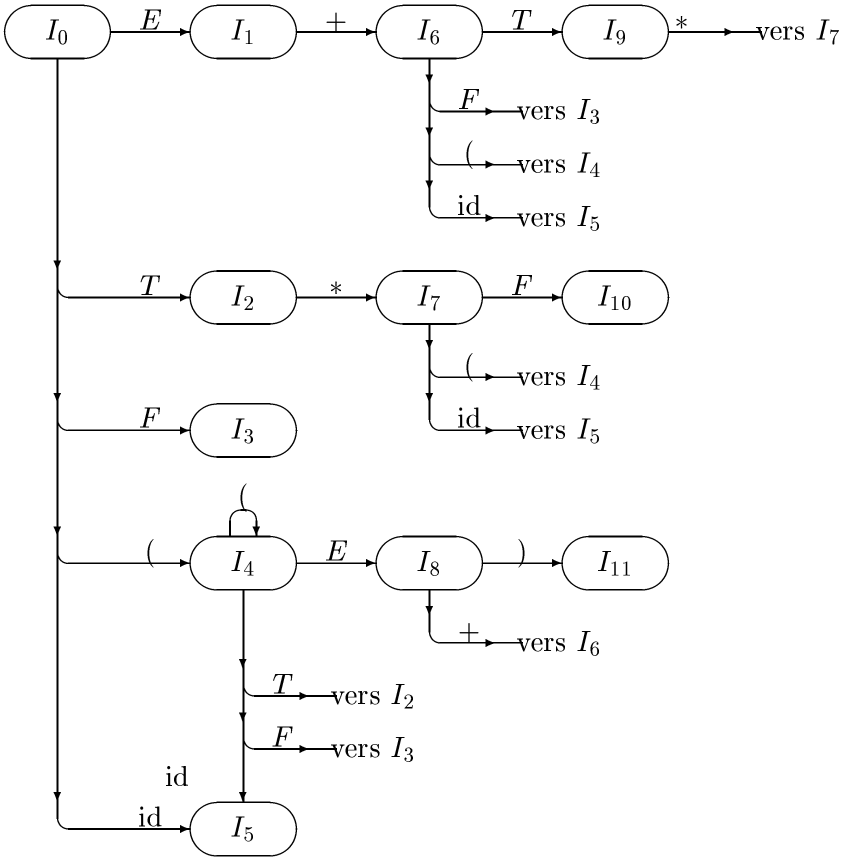


   L'état 4 est terminal pour reconnaître le langage T; pour reconnaître U, les états terminaux sont au contraire $0, 1, 2, 3$ (attention : ce raisonnement n'est correct que pour un automate déterministe).

   Cet automate est minimal, car le nombre d'états nécessaires à une reconnaissance de motif est égal au nombre de préfixes de ce motif. On peut aussi le vérifier rapidement avec l'algorithme usuel.

2 Analyse syntaxique

On rappelle l'automate utilisé pour une analyse ascendante de G :



1. $a * (b + c)$ est une expression reconnue par G et non reconnue par G_1. Le langage reconnu par G_1 est constitué des expressions dans lesquelles l'opérande droit d'une multiplication est toujours un identificateur (jamais une expression entre parenthèses).

   Remarque : les multiplications consécutives sont autorisées ; par exemple $a * b * c$ est une expression reconnue par G_1.

   Différences entre l'automate associé à G_1 et celui (ci-dessus) associé à G :

   - l'état I_3 n'existe plus, de même que les transitions étiquetées F ;
   - l'état I_7 est source d'une seule transition, vers I_{10}, étiquetée *id*.

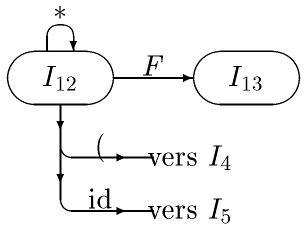
   Les conflits surviennent, pour G, dans les états I_2 (réduire T en E ou empiler $*$) et I_9 (réduire $E + T$ en E ou empiler $*$) ; ce sont exactement les mêmes pour G_1, qui comporte donc autant de conflits que G.

2. G_2 n'est probablement pas ambiguë, car l'utilisation du symbole $*$ pour préfixer une expression est courante ... en C (opérateur d'indirection) !

   L'automate associé à G_2 comporte deux états supplémentaires :

- $I_{12} = \{\ F \to *.F,\ F \to .*F,\ F \to .(E),\ F \to .\mathrm{id}\ \}$
- $I_{13} = \{\ F \to *F.\ \}$.

Des transitions étiquetées $*$ vont de I_0, I_4, I_6, I_7 vers I_{12}, et I_{12} est source des transitions suivantes :



En I_{12} la seule action possible est d'empiler le token suivant, et en I_{13} la seule action possible est la réduction $*F \to F$. Donc l'analyse ascendante de G_2 ne comporte pas de nouveau conflit, et la résolution de ceux de G est identique, car Suiv(E) est inchangé, et ne contient pas $*$. G_2 est donc SLR(1), comme G.

3. G_3 n'est pas ambiguë : le symbole $*$ est un symbole de multiplication s'il est suivi d'un identificateur ou d'une parenthèse ouvrante, et c'est un opérateur postfixé sinon (ie. s'il est suivi des symboles $+, *$, d'une parenthèse fermante, ou de la marque de fin de chaîne).

   Par contre, G_3 n'est pas SLR(1) : des conflits apparaissent en I_3 (réduire F en T ou empiler $*$) et I_{10} (réduire $T * F$ en T ou empiler $*$), et ils ne peuvent être résolus, car Suiv(T) contient le symbole $*$.

   G_3 n'est pas LR(1), car dans l'analyse des expressions $a * b$ et $a * +b$, après lecture de a et réduction de id en F, on est en I_3, le token suivant est $*$, et pourtant la suite de l'analyse diverge :

   - dans le cas de $a * b$, il faut réduire F en T avant d'empiler $*$;
   - dans le cas de $a * +b$, il faut empiler $*$, puis réduire $F*$ en F.

   Le choix de l'action dépend du *deuxième* token courant (id dans le premier cas, $+$ dans le second). On dit que G_3 est LR(2).

4. Comparé à l'analyseur ascendant pour la grammaire (ambiguë) équivalente à G, celui de G_4 présente deux nouveaux états :

   - $I_{20} = \{\ E \to *.E,\ E \to .E{+}E,\ E \to .E{*}E,\ E \to .*E,\ E \to .(E),\ E \to .\mathrm{id}\ \}$
   - $I_{21} = \{\ E \to *E.,\ E \to E. + E,\ E \to E. * E\ \}$

   Le conflit en I_{21} (réduire $*E$ en E ou empiler un opérateur) doit être toujours résolu en faveur de la réduction, pour garantir la priorité de l'opérateur unaire préfixé (indirection en C).

3 Attributs

1. On définit un attribut qui, outre les champs habituels nécessaires à la construction de l'arbre (opérateur ou identificateur, fils gauche, fils droit), contient deux nouveaux champs, *numerique* pour indiquer si le nœud est une constante numérique (et donc une feuille), et *valeur* qui contient la valeur éventuelle de cette constante.

   Le schéma de calcul des attributs est alors le suivant :

```
E  : E '+' T  { if ($1.numerique && $1.valeur == 0)
                  $$ = $3;
                else if ($3.numerique && $3.valeur == 0)
                  $$ = $1;
                else if ($1.numerique && $3.numerique)
                  /* $$ = $1 implicite */
                  $1.valeur = $1.valeur + $3.valeur;
                else
                  $$ = noeud ('+', $1, $3); }
   | T
   ;
T  : T '*' F  { if ($1.numerique && $1.valeur == 1)
                  $$ = $3;
                else if ($3.numerique && $3.valeur == 1)
                  $$ = $1;
                else if ($1.numerique && $3.numerique)
                  /* $$ = $1 implicite */
                  $1.valeur = $1.valeur * $3.valeur;
                else
                  $$ = noeud ('*', $1, $3); }
   | F
   ;
F  : '(' E ')'  { $$ = $2; }
   | ID
   | NUM
   ;
```

   en supposant que l'analyseur lexical place dans *yylval* (un pointeur sur) une feuille, avec les champs *numerique* et *valeur* convenablement remplis.

   Cette solution ne produit pas d'expression simplifiée pour des expressions telles que $2 * x * 3$, car l'arbre syntaxique ne contient pas la sous-expression $2 * 3$.