Lic. Info. 3ème année — SLA6IF05 - Langages algébriques
Université d'Orléans 2019-2020 — Wadoud BOUSDIRA

Devoir de programmation.

Réalisation d'un analyseur syntaxique pour le langage ALGO1
À rendre le 15/5/2020

Ce devoir n'est pas uniquement un devoir de programmation. Vous y trouverez aussi des questions auxquelles vous répondrez dans un document séparé qui fera partie de l'archive que vous déposerez sur Celene. Ces questions seront précédées du symbole ★ et seront évaluées indépendamment du travail de développement demandé.

On souhaite développer un analyseur lexical et syntaxique pour une version simplifiée du langage algorithmique utilisé en 1ère année de licence, qu'on appelle `ALGO1`.

1 Spécification lexicale :

- *Valeurs et types :* on considère uniquement les variables et les constantes de type entier ou tableau d'entiers, et les constantes booléennes `true` et `false`. Toutes les informations sont de type entier ou tableau d'entiers et toutes les bexpressions sont de type booléen.
- L'identificateur appartient au langage exprimé par l'expression régulière $[a..z]\,([a..z] \mid [0..9])^*$. [1]

Dans cette partie, il vous est demandé de

1. ★ définir l'automate d'états fini minimal qui reconnaît les entiers correctement écrits, i.e. les mots de la forme $0 \mid (- \mid \epsilon)\,([1..9]\,[0..9]^*)$.
2. ★ définir l'automate d'états fini minimal qui reconnaît les identificateurs. Cet automate reconnaîtra entre autres les mots clé du langage répertoriés ci-dessous :

`program`	`end`	`break`	`begin`
`while`	`for`	`if`	`do`
`true`	`false`	`from`	`not`
`then`	`else`	`and`	`or`

3. coder la reconnaissance lexicale par l'automate de 1 et l'automate de 2. Il vous est demandé de préciser la structure de données choisie pour représenter un automate.
4. coder une fonction qui prend en entrée un programme écrit en `ALGO1` et qui retourne en sortie une chaîne de caractères correspondant au programme initial dans lequel tous les identificateurs, à l'exception des mots clé, (respectivement tous les entiers) ont été remplacés par le token `ident` (resp. par le token `entier`). On se charge aussi de supprimer les espaces non significatifs et les sauts de ligne.
   Par exemple, pour **Exemple 1** de l'annexe, la fonction demandée retourne la chaîne

1. La notation $[a..z]$ représente un caractère quelconque compris entre $'a'$ et $'z'$; même chose pour [0..9] et [1..9].

`program ident begin ident <- ident;ident <- ident;ident <- ident end.`

- On ne s'occupera pas de créer une table des symboles qui a toute son utilité pour les analyses qui suivent la reconnaissance syntaxique, mais qu'on ne code pas ici.
- C'est cette étape qui distingue les mots clé des autres identificateurs.

2 Spécification syntaxique :

Cette partie consiste à développer un analyseur syntaxique fondé sur l'approche LL(1). La grammaire $\mathcal{G}$ qui vous est donnée n'est pas LL(1). Il s'agira dans un premier temps de la transformer pour la rendre LL(1).

a) Les non-terminaux commencent par une majuscule, et les terminaux sont en minuscules. `S` est l'axiome de la grammaire.

b) Pour rappel, le terminal `ident` est utilisé pour représenter un identificateur et le terminal `entier` représente une constante entière.

c) Les opérateurs arithmétiques binaires ont les priorités usuelles, i.e. l'addition et la soustraction sont moins prioritaires que la multiplication et la division. D'une part, l'addition et la soustraction sont de même priorité, et d'autre part, la multiplication et la division ont la même priorité.

d) Les opérateurs logiques ont les priorités usuelles, i.e. la disjonction est moins prioritaire que la conjonction, elle-même moins prioritaire que la négation.

e) Les opérateurs arithmétiques et logiques sont tous associatifs à gauche.

2.1 Mise sous forme LL(1)

1. ★ Expliquer pourquoi $\mathcal{G}$ n'est pas LL(1).
2. ★ Proposer une grammaire LL(1) $\mathcal{G}'$ équivalente à $\mathcal{G}$. Préciser toutes les vérifications nécessaires pour justifier que $\mathcal{G}'$ est LL(1).

$\underline{\mathcal{G}}$:

`S`	→	`program` ident begin LI `end`.
`LI`	→	`I` ; `LI`
	|	`I`
`I`	→	`Affectation`
	|	`While`
	|	`For`
	|	`If`
	|	`break`
`Affectation`	→	ident <- `Expression`
	|	ident <- `ValBool`
`While`	→	while `BExpression` do `LI` end
`For`	→	for ident from `Valeur` to `Valeur` do `LI` end
`If`	→	if `BExpression` then `LI` else `LI` end
`ValBool`	→	`true` | `false`
`BExpression`	→	`ValBool`
	|	not `BExpression`
	|	`BExpression` or `BExpression`
	|	`BExpression` and `BExpression`
	|	`Condition`
	|	`(BExpression)`
`Expression`	→	`Expression OpArith Expression`
	|	(`Expression`)
	|	`VarNum`
`OpArith`	→	+ | - | * | /
`VarNum`	→	ident | ident [`Expression`] | entier
`Valeur`	→	ident | entier
`Condition`	→	`Expression OpRel Expression`
`OpRel`	→	<= | < | > | >= | = | !=

2.2 Calcul des ensembles Premier et Suivant

Dans cette partie, on demande de coder les algorithmes donnés en cours pour construire les ensembles Premier des alternatives de chaque non-terminal et les ensembles Suivant de chaque non-terminal pour une grammaire $\mathcal{G}$ quelconque définie par ses quatre composantes,

1. l'ensemble des terminaux Σ,
2. l'ensemble des non-terminaux N,
3. l'axiome S
4. l'ensemble des règles de production $\mathcal{P}$.

Choisir une structure de données appropriée pour stocker une grammaire, que vous prendrez soin de justifier. Vous devrez pour chacun des algorithmes, prévoir un affichage des ensembles calculés.

Appliquer vos algorithmes à la grammaire $\mathcal{G}'$ de `ALGO1`. Assurez-vous que les résultats calculés sont ceux attendus.

2.3 Construction de la table d'analyse LL(1)

Une fois que les ensembles Premier et Suivant sont calculés, on est en mesure de construire la table d'analyse. Étant donnée une grammaire $\mathcal{G}$, on demande de coder l'algorithme de construction de la table d'analyse correspondante, en utilisant les ensembles Premier et Suivant calculés dans la section précédente.

On demande de prévoir un affichage de la table d'analyse.

Appliquer votre algorithme pour construire la table d'analyse de la grammaire $\mathcal{G}'$ qui reconnaît le langage `ALGO1`.

2.4 Analyse de chaînes

Cette partie consiste à programmer l'analyse elle-même. Avec la table d'analyse de la grammaire, la fonction d'analyse prend en entrée une chaîne (se terminant par le symbole spécial \$) et elle retourne une réponse binaire, indiquant si la chaîne est syntaxiquement correcte conformément aux règles de la grammaire ou non.

Appliquer votre fonction d'analyse à la grammaire $\mathcal{G}'$.

3 Interface de l'analyseur syntaxique

Cette dernière partie consiste à coder un menu avec deux options :

1. une option qui lit une grammaire G dont on suppose qu'elle est LL(1) et qui retourne la table d'analyse de G ;

   Pour cette option, prévoir une interaction suffisante avec l'utilisateur pour déterminer les terminaux et les non-terminaux (entre autres l'axiome) de la grammaire.

   On devra pouvoir afficher le résultat des différentes parties de la section 2.

   Vous pourrez tester cette partie avec les grammaires vues en cours et en TD. Pour cela, il faudra avoir la possibilité de lire une grammaire à partir d'un fichier.
2. une option qui lit un programme écrit en `ALGO1` et qui l'analyse.

   Vous trouverez en annexe des exemples de programmes écrits en `ALGO1`.

4 Travail demandé

Le travail de programmation est à faire individuellement. Il devra être codé en Java. Le rendu sera sous forme d'archive "**(.zip)**" comprenant

1. les fichiers source .java, suffisamment commentés.
2. l'exécutable
3. un guide d'utilisation du projet, expliquant comment l'exécuter comprenant une partie expliquant vos choix de développement.
4. un document texte (.pdf, ou .doc) pour les réponses aux questions marquées du symbole ★.
5. Des cas de tests accompagnés des résultats attendus et retournés.

Un dépôt sera créé pour ce devoir.

5 Annexe :

Exemple 1

```
program p1
begin
   z <- x ;
   x <- y ;
   y <- z
end.
```

Exemple 2

```
program p2
begin
   z <- t[i] ;
   t[i] <- t[j] ;
   t[j] <- z
end.
```

Exemple 3

```
program p3
begin
   x <- 100;
   while 0 < x do
     x <- x -1;
   end
end.
```

Exemple 4

```
program p4
begin
   i <- 100 ;
   z <- 1;
   while i >= 0 do
     z <- z*t[i]
   end
end .
```

Exemple 5

```
program p5
begin
   r <- 1 ;
   for i from 2 to n do
     r <- r * i
   end
end .
```

Exemple 6

```
program p6
begin
   i <- 0 ; r <- true;
   while i < n-1 do
     if t[i] > t[i+1] then
      r <- false ;
      break
     else r <- true
     end ;
     i <- i+1
   end
 end.
```