COMPILATION TD n° 4

DEVAN SOHIER

`Lecs` est un programme créant un analyseur lexical à partir d'expressions régulières associées à des instructions. L'analyseur lexical exécute ces instructions lorsqu'une séquence de caractère est identifiée avec l'expression régulière correspondante.

Lorsque plusieurs expressions régulières peuvent être analysées, c'est celle qui permet d'analyser la plus longue chaîne qui est choisie. Lorsqu'une même chaîne peut être reconnue par plusieurs expressions régulières, c'est la première dans l'ordre d'écriture qui est choisie. Les chaînes non-reconnues sont reproduites en sortie. Cette chaîne reconnue est stockée dans la variable `texte`.

Ainsi, l'analyseur lexical produit par le programme ci-dessous devra transformer la séquence `for 0a111 fort9a` en `BOUCLE NOMBRE VARIABLE VARIABLE`

```
for {écrire BOUCLE}
[a-z][a-z0-9]* {écrire VARIABLE}
[0-9]+ {écrire NOMBRE}
```

`Lecs` construit à partir des expressions régulières un automate non déterministe par la méthode de numérotation reconnaissant la fermeture itérative de leur disjonction, et associe à chaque état terminal les instructions correspondant à l'expression régulière

(1) Pour chacune des trois expressions régulières du programme, écrire par la méthode des automates standards un automate fini reconnaissant le même langage.

(2) Toujours par la méthode des automates standards, en faire la disjonction.

(3) Déterminiser cet automate, et pour chaque état accepteur, indiquer l'action du programme qu'il faudra effectuer en prenant en compte les règles de priorité.

(4) Minimiser l'automate obtenu (vous prendrez garde au fait que le comportement de l'automate diverge selon l'état terminal qu'il atteint, et que l'on ne peut donc pas tous les rassembler).

On obtient ainsi un automate minimal reconnaissant un lexème. On souhaite maintenant reconnaître une séquence de lexèmes et exécuter les actions associées.

(5) Après avoir retiré tout état mort, pour chaque état terminal, et pour chaque transition depuis l'état initial, ajoutez cette même transition

s'il n'en existe aucune de même étiquette depuis l'état terminal ; rendez l'état initial accepteur (**NB :** remarquez la similitude avec la *"mise à l'étoile"* par la méthode des automates standards ; on l'adapte afin de conserver le déterminisme de l'automate, et de garantir que l'on reconnaît le plus long préfixe de la chaîne donnée).

(6) Les caractères non-reconnus dans un état terminal sont reproduits sur la sortie standard (ils tiennent lieu de séparateurs). Ajoutez une transition depuis tout état accepteur sur tout caractère non-reconnu vers l'état initial.

(7) Ecrire (en pseudo code) l'analyseur lexical obtenu : à chaque transition ajoutée dans les questions 5 et 6 est associée l'action de l'état dont elle est issue ; à chaque transition ajoutée à la question 6 est également associée l'action d'écrire sur la sortie standard le caractère lu.

(8) Même questions avec le programme ci-dessous :

```
[0-9]+ {empiler(p, valeur(texte))}
+ {i:=depiler(p); j=depiler(p); empiler (i+j)}
* {i:=depiler(p); j=depiler(p); empiler (i*j)}
- {i:=depiler(p); j=depiler(p); empiler (i-j)}
/ {i:=depiler(p); j=depiler(p); empiler (i/j)}
-- {i:=depiler(p); empiler (-i)}
' '{}
\n {i:=depiler(p); si(est_vide(p))écrire(i);sinon erreur !!!}
. {erreur !!!}
```

(9) Que fait l'analyseur ainsi construit ? Remarquez que l'utilisation d'une pile permet à ce programme de détecter des erreurs syntaxiques.