Marion SAROUL
Paola CECCATO
SICOM - B

le 30 septembre 2012

Projet assembleur : rapport sur l'analyse lexicale

I Objectifs de la première étape
Analyse lexicale d'un fichier texte assembleur

Dans cette première étape nous devions effectuer l'analyse lexicale d'un fichier texte assembleur. Cette partie du projet était donc divisée en plusieurs étapes : Nous devions dans un premier temps lire le fichier assembleur ligne par ligne, puis canoniser chacune de ses lignes afin de pouvoir par la suite les diviser en « mot » que l'on appellera lexème. Chacun de ses lexèmes devaient être divisés en différents types afin de faciliter par la suite leur analyse grammaticale et sémantique. Notre fonction doit pour finir renvoyer la liste de chacun de ces lexèmes associés à leur type .

II Nos fonctions

Notre programme principal s'intitule « étape1.c », il fait appelle à 4 fonctions principales contenues dans « auto_nombre.c » et « liste_chainee.c ».

1) Canoniser :

- **Objectifs de la fonction**

Cette fonction a pour objectif de "mettre en forme" la ligne que nous allons analyser. C'est un travail préliminaire indispensable pour pouvoir par la suite couper la ligne en lexèmes. Pour ceci, il faut isoler les caractères qui sont "d'eux-mêmes" un lexème. Ces caractères sont les suivants :

- la virgule (',')
- les deux points (':')
- la parenthèse ouvrante ('(')
- la parenthèse fermante (')')
- le plus ('+')
- le moins ('-')

La tabulation ('\t') doit quant à elle être remplacée par un espace ('\n').
Le reste de la ligne reste inchangé.

Note : On ne supprimera pas les espaces multiples car cela n'est pas nécessaire pour la fonction coupe_ligne.

- **Conception de la fonction**

Pour ceci nous avons créé une fonction « canoniser » qui prend en paramètre d'entrée une ligne et qui renvoie une chaîne de caractère.

Afin de mettre en forme la ligne placée en paramètre, nous avons crée une chaîne de 1 000 caractère initialisée à 0. Cette méthode a été choisi car on considère que les lignes à traiter ne dépasserons pas ce nombre de caractères. Si nécessaire, nous pourrons par la suite créer une chaîne plus grande. Cela nous évite en effet de faire une allocation dynamique pour chaque ligne ce qui rend le programme moins "lourd".

On balaie ensuite la ligne caractère par caractère et on traite tous les cas particuliers cités précédemment .

Une petite astuce réside dans le cas général (ie tous les autres caractères). Si on détecte un caractère "normal", prenons par exemple la lettre 'p'. On ajoute dans un premier temps un 0 à la suite de la chaîne (ie en s [strlen(s)+1] avec s le nom de la chaîne traitée). Dans un second temps, on vient placer la lettre 'p' avant ce 0 (ie en s [strlen(s)]). Cette astuce permet de toujours avoir un 0 en fin de chaîne pour que les conditions de la boucle restent valables pour les caractères suivants.

On ne retourne pas la chaîne traitée (s) mais sa duplication. En effet une fois que la fonction a traité la ligne la mémoire allouée pour s est supprimée.

2) Coupe-ligne :

- **Objectifs de la fonction :**

Une fois la ligne canonisée, elle a été mise en forme pour l'analyse. La deuxième étape, permet de couper la ligne en "mot" grâce à la fonction coupe_ligne. Nous conviendrons pour cela que le séparateur de lexèmes est l'espace.

- **Conception de la fonction :**

Coupe_ligne prend en paramètres d'entrée, la chaîne de caractères à découper et un tableau de "mot" (chaîne de caractère) qui contiendra les mots de la lignes qui ont été découpés . Cette fonction renvoie le nombre de "mots" qu'il y a dans le tableau.

Pour ceci, coupe_ligne lit la chaîne de caractère, caractère par caractère, et la coupe en lexèmes lorsqu'elle rencontre un espace. Elle place ensuite les lexèmes, les uns après les autres dans un tableau de mots. A chaque fin de ligne elle place le caractère de fin de ligne « \n » dans le ième+1 case du tableau.

Dans un second temps, elle balaie le tableau et ajoute le caractère espace à chaque fin de mot pour faciliter la suite de l'analyse lexicale.

Note : Lorsque la fonction rencontre le caractère '#' cela signifie que le reste de la ligne est un commentaire. Dans ce cas elle ne doit plus considérer les espaces comme des séparateurs et doit placer toute la ligne à partir du # dans une même case du tableau. Pour ceci, nous avons dupliqué la chaîne entrée en paramètres (dans save). Ainsi la duplication aura allouer juste la taille nécessaire. Grace à la fonction strstr, on détecte le caractère '#' et on pointe vers son adresse (psave). En balayant le tableau, dans le cas où le # sera détecter, on pourra ainsi aisément copier toute la suite de caractère dans la ième case du tableau.

3) Classifier :

- **Objectifs de la fonction :**

Cette fonction a pour but de classer les différents "mots" en lexème. Cette étape permettra de faciliter par la suite les analyses grammaticales et sémantiques.

- **Conception de la fonction :**

Pour classifier nos lexèmes nous nous sommes basées sur la fonction auto_nombre déjà créée . Nous avons ajouté les Lexèmes : NL ,SYMBOLE , DEUX_PTS , VIRGULE , REGISTRE , PARENTHESE_OUVRANTE, PARENTHESE_FERMANTE, COMMENT, DIRECTIVE, PLUS et MOINS .

Pour les lexèmes tel que : virgule (ou deux points, retour à la ligne, parenthèses, plus et moins), formés d'un seul caractère, nous testons ce caractère dans le cas INIT puis nous donnons S=VIRGULE. Le deuxième caractère est donc testé dans le cas VIRGULE, dans ce « cas » on test si virgule est bien suivi d'un espace, si c'est le cas, la fonction renvoie le numéro associé à VIRGULE. Sinon la fonction affiche une erreur.

Pour les lexèmes qui sont formés de plusieurs caractères différents tels que les registres ($6) ou les directives (.debut), nous avons testé le premier caractère dans le cas INIT. Puis dans chacun des cas testés si le second caractère est bien un chiffre (registre) ou une lettre (directive). Dans le cas contraire, un message d'erreur indiquant le caractère erroné et le "mot" auquel il appartient est envoyé à l'utilisateur.

4) Enfiler :

- **Objectif de la fonction :**

Une fois chaque "mot" classé il nous faut les ranger dans une liste qui prend en paramètre le mot et son type de lexème associé. Nous avons travaillé dans un premier temps sur une fonction "ajout_queue" mais nous nous sommes aperçu rapidement que cela demandait à la fonction de parcourir la liste entièrement à chaque fois. Nous avons donc décidé de travailler sur une file.

- **Conception de la fonction :**

Dans cette étape il était important de pouvoir avoir accès à la fin de notre liste sans avoir à la parcourir à chaque fois. Nous avons donc traité notre liste comme une file qui se boucle sur elle même (c'est à dire que le dernier pointeur pointe sur la première cellule de la liste)

III Tests

1) Test de la fonction canoniser

- **Objectifs du test :** On souhaite regarder si lorsque la fonction détecte un caractère qui de lui même est un lexème (voir partie I du rapport), un espace est ajouté avant et après le caractère. De même on vérifie que les chaînes de lettres/nombres/$/# sont bien restées inchangées.

- **Résultat du test :** La fonction canonise bien notre phrase, elle met des espaces de chaque coté des caractères ‘,’ ou ‘+’ et ne sépare pas “$6” . Nous n’avons à ce jour par détecté d’erreurs .

2) Test de la fonction coupe_ligne

- **Objectifs du test :** En balayant le tableau de mots on affiche pour chaque i, mot[i]. On vérifie ainsi, en passant en paramètre une ligne déjà canonisée les points suivants :
  - Après un ‘#’ on ne coupe pas lorsqu’un espace est détecté : c'est un commentaire.
  - Un espace a bien été ajouté après chaque mot (on print le caractère ‘s’ directement après mot[i] pour mettre l’espace en évidence.

- **Résultats du test :** Noter fonction nous renvoie bien un tableau de mot rempli par les différents lexèmes de la phrase. Un double espace équivaut à un espace simple . La aussi nous n’avons à ce jour par détecté d’erreurs .

3) Test de la fonction classifier

- **Objectifs du test :** Dans ce teste nous vérifions que chaque lexème est bien identifié et classé. Nous vérifions également qu’un message d’erreur est bien envoyé si le mot ne peut pas être classé dans un type de lexème .

- **Résultats du test :** Notre fonction nous renvoie bien chaque type de lexème associé au bon mot . Un message d’erreur est également bien affiché .

4) Test de la fonction enfiler

- **Objectifs du test :** Insérer deux chaînes de caractères à la fin d’une liste et visualiser cette liste.

- **Résultats du test :** La fonction affiche bien la liste avec les deux chaînes de caractères dans un premier temps puis les chaînes de caractères numérotées, ce qui nous permet de vérifier que l'on a bien l'ordre d'insertion souhaité.

5) Test général :

- **Objectif :** Notre fichier assembleur comportera deux parties, une partie sans erreur et une partie avec erreur (pour cela nous mettrons en commentaire dans notre main l'arrêt de notre programme en cas d'erreur). Nous pourrons ainsi observer quelles erreurs sont détectées et le comportement de notre programme face à celles ci .

- **Résultat du test :** La première partie de notre texte assembleur est bien traitée . Chaque mot est associé au bon lexème et la liste s'est bien affichée . Les erreurs détectées par la suite sont bien celle que nous attendions. Par exemple : le programme n'acceptera pas les espaces après les '.' (". directives") ou après les '$' ("$ ").

- **Problèmes de codes mis en évidence grâce aux tests :** Durant ce test nous nous sommes aperçus que des chaînes de caractères étaient ajoutées après un commentaire. Nous avions en effet oublié de traiter le caractère fin de ligne dans le cas d'un commentaire.

IV Conclusion sur l'analyse syntaxique, objectifs et améliorations possibles

Notre programme lit le texte assembleur, le découpe en lignes puis en mots et classe chacun de ces mots en lexèmes. Il nous renvoie une liste comportant les mot et lexèmes associés. Il renvoie également un message d'erreur dans la fonction classifie en indiquant le caractère qui pose problème et le mot auquel il appartient.

Nous avons donc atteint tous les objectifs que nous nous étions fixés. Nous avons également eu le temps de modifier le fonctionnement de notre programme face, par exemple, à plusieurs sauts de lignes dans le fichier assembleur.

Améliorations possibles :

Il nous reste cependant des modifications possibles à apporter :

- Il ne sera en effet pas possible pour l'utilisateur, dans le cas SYMBOLE , de mettre un autre caractère que des lettres ou des chiffres donc toutes ponctuations renverra un message d'erreur .
- Une ligne ne pourra pas dépasser mille mots puisque nous avons créé un tableau de mot de taille mille. Dans la suite de notre projet on pourra allouer dynamiquement la taille du tableau si nécessaire.
- Il pourrait être intéressant de faire apparaître le numéro de ligne en cas d'erreur. La correction pour l'utilisateur serait ainsi plus facile.