Polytech Nice Sophia
SI4 / 2021-2022
Erick Gallesio

Compilation

Feuille 1

Analyse Lexicale

Lex / Flex

> *Dans cette feuille de TD, nous allons construire des analyseurs en utilisant le générateur d'analyseurs **lex** (ou **flex**).*
>
> ***Attention:***
> *Les premiers exercices de cette feuille ont déjà été vus dans le cours (optionnel) de LFA en SI3.*
> *Ils peuvent être sautés par ceux qui avaient choisi ce cours l'an dernier (et qui se souviennent du contenu des Tds de LFA).*

Avant de commencer

Copiez le fichier Makefile dans votre répertoire de travail pour réaliser les exercices demandés.

Ce fichier Makefile permet de transformer tous les fichiers suffixés par `'.l'` en des executables (aprés avoir compilé l'automate en C produit pas la commande *lex*). Ainsi, si vous avez un fichier `exo.l` et que vous lancez la commande Unix `make`, vous obtenez (si tout va bien) un exécutable de nom `exo` que vous pouvez exécuter classiquement sous le shell avec `./exo`.

Noter que le fichier `.c` n'est pas conservé. Pour les curieux, vous pouvez produire le fichier `.c` avec la commande `make exo.c`.

Un exemple de session est montré ci-dessous:

```
$ ls
Makefile  catlex.l
$ make
lex -o catlex.c catlex.l
gcc -Wall -Wextra -std=gnu99 -Wno-unused-function   -c -o catlex.o catlex.c
gcc   catlex.o   -o catlex
rm catlex.o catlex.c
$
```

Le source du programme `catlex.l` utilisé ici est assez minimal; il permet de simuler la commande `cat` de `Unix` :

```
 /* La commande cat en lex */
%%

%%
int main()  { return yylex(); }
int yywrap() { return 1; }
```

Notes:

- Par défaut, le code produit par *lex* utilise (appelle) une fonction nommée `yywrap`. Cette fonction est appelée quand on arrive en fin de fichier; il faut donc qu'elle soit présente dans votre programme.
- On peut éviter la production de l'appel à la fonction `yywrap`
    - soit avec l'option `%option noyywrap` dans le fichier lex,
    - soit avec l'option `--noyywrap` de `flex`.

  Malheureusement, ces deux solutions ne marchent pas toujours (ou mal) suivant la version de *lex/flex* que vous utilisez. Dans le doute, vous pouvez faire comme ci-dessus pour éviter la génération de message d'alertes.

1 Commande upper

Écrire la commande `upper` qui recopie en majuscules les caractères du fichier standard d'entrée sur le fichier standard de sortie. Vous pouvez partir du programme `catlex.l` précédent pour construire votre programme.

Rappels:

- En lex, le texte reconnu par un règle est contenu dans la chaîne `yytext` ;
- En C, la fonction `toupper(c)` permet de mettre en majuscule le caractère contenu dans `c`. Cette fonction est définie dans le fichier d'en-têtes `<ctype.h>`.

2 Commande Unix wc

Écrire un clone de la commande *wc*. On rappelle que cette commande affiche le nombre de lignes, mots et caractères des fichiers qui lui sont passés en paramètre. Dans *wc*, les seuls séparateurs de mots sont les caractères espace, tabulation ou newline.

Pour simplifier, votre commande travaillera uniquement sur le fichier standard d'entrée et n'acceptera pas d'option.

Vérifiez que votre commande renvoie bien les même résultats que la commande standard.

3 Analyse des chaînes de caractères C

En C, les chaînes de caractères sont délimitées par le caractère guillemet (`'"'`). Pour pouvoir entrer un caractère `'"'` ou un caractère `'\'` dans une chaîne, il faut le précéder du caractère `'\'`. Ainsi, les chaînes suivantes:

```
"I'm a string"
"Another string with embedded \"quotes\""
"and another one with a '\\' or even \"\\\" !!!"
```

sont des chaînes correctes en C. En fait, le caractère `'\'` permet aussi d'entrer des caractères spéciaux (newline, tab, bell, …) ou des caractères exprimés en octal ou en hexadécimal.

Exemples:

```
"Printed text on\n2 lines"
"The TAB character: '\t' or '\011' or '\x9'."
"A string spanning \
 2 lines"
```

Écrire un programme permettant de mettre en majuscules les chaînes de caractères d'un programme C (et seulement les chaînes de caractères).

On écrira deux versions de ce programme:

- la première version utilisera une expression régulière, et
- la seconde utilisera la notion de *contexte gauche* vue en cours.

Pour tester votre programme, vous pourrez prendre le fichier input1.c.

4 Analyse des commentaires C

Écrire un programme qui supprime les commentaires d'un programme C. Votre programme ne devra bien sûr pas supprimer les commentaires qui pourraient se trouver à l'intérieur d'une chaîne.

On rappelle qu'il existe deux formes de commentaires en C:

- les commentaires délimités par les séquences `'/*'` et `'*/'` qui peuvent se répartir sur plusieurs lignes.
- les commentaires qui commencent par la séquence `'//'` et qui se terminent en fin de ligne.

Première version:

La première version devra reconnaître les deux formes de commentaires de C. Utilisez des contextes gauches pour la reconnaissance des commentaires. Bien sûr, le texte produit par votre programme devra toujours pourvoir être compilé.

Là encore, pour tester votre programme, vous pourrez prendre les fichiers input1.c et input2.c (attention, il y a un piège).

Deuxième version:

Étendez votre programme pour qu'il accepte des commentaires emboîtés (ce qui n'est pas admis en C standard).

Pour tester votre programme, vous pouvez utiliser le fichier input3.c.

Troisième version:

Ajouter un message d'erreur si vous rencontrez la fin de fichier lors de la lecture d'un commentaire. La fin de fichier est dénotée en *lex* par la séquence `<<EOF>>` dans la partie modèle d'une règle.

5 Reconnaissances de nombres

À l'aide d'expressions régulières, on veut remplacer les nombres d'un fichier par la chaîne **NOMBRE**.

1. Trouver l'expression pour reconnaître des nombres entiers (signés ou non) comme, par exemple, *10*, *-7*, *+42*.
2. Modifier votre expression régulière pour reconnaître des nombres flottants comme, par exemple, *10*, *-7*, *+42*, *+10.21*, *-10.21*, *+.21*, *.21*
3. Modifier votre expression régulière pour reconnaître des nombres flottants avec un exposant éventuel comme, par exemple, *10*, *-7*, *+42*, *+10.21*, *-10.21*, *+.21*, *.21*, *-.4e18*, *10.3E-17*.

6 Évaluation de nombres C

En utilisant des start conditions, écrire un analyseur des nombres entiers C qui calcule leur valeur au fur et à mesure qu'on en reconnaît ses chiffres:

Un exemple d'exécution du programme est donné ci-dessous:

```
$ nombres
1234
Decimal: 1234
01234
Octal: 668
0x1234
Hexadecimal: 4660
```

7 Un analyseur lexical pour une calculatrice

Dans cet exercice, on va construire l'analyseur lexical pour une calculatrice simple. Cette calculatrice travaille sur des nombres entiers et dispose des 4 opérations classiques (`'+'` , `'-'` , `'*'` , `'/'`). Elle accepte aussi des expressions parenthésées et des variables.

Un exemple de session avec cette calculatrice est donné ci-dessous:

```
$ calc
[1] x=y=10
==> 10
[2] x*y+2
==> 102
[3] (x+y)*3
==> 60
[4] foo=(x+y)*3
==> 60
[5] foo
==> 60
[6] bar
*** Error: variable 'bar' non initialisée
==> 0
[7] Bye
```

Le programme calculatrice est formé de deux parties:

- un analyseur lexical (que vous devez écrire)
- un analyseur syntaxique (qui vous est fourni)

L'analyseur syntaxique qui vous est distribué suppose que l'analyseur renvoie des unités syntaxiques dont les noms sont:

- **NUMBER** pour les nombres (entiers seulement)
- **IDENT** pour les noms de variables
- **PLUS, MINUS, MULT, DIV** pour les 4 opérateurs
- **OPEN** et **CLOSE** pour les parenthèses
- **EOL** quand on rencontre une fin de ligne
- **EQUAL** pour l'affectation

Par ailleurs, l'analyseur lexical et syntaxique peuvent communiquer au travers la variable `yylval` lorsqu'on lit un entier ou une variable:

```
union {
  char *var;   // nom de variable
  int val;     // valeur d'un entier
} yylval;
```

Comme cette variable est déclarée par l'analyseur syntaxique, vous n'avez pas à le faire.

Pour construire votre calculatrice, vous devez récupérer les fichiers de l'analyseur syntaxique:

- calc.h qui définit les noms d'unités syntaxiques et la variable `yylval` , et
- calc.c qui contient l'analyseur syntaxique et le programme principal (ce fichier a été généré par le générateur d'analyseurs `bison`).

Une calculatrice étendue

Modifier votre analyseur lexical pour:

- accepter que les nombres puissent être exprimés en hexadécimal ou en octal
- accepter des commentaires (comme en Python de `#` jusqu'à la fin de la ligne)
- accepter la commande `quit` pour arrêter la calculatrice
- permettre de lancer des commande Unix si la ligne commence par un `!` . Ainsi, la ligne `!pwd` affiche le répertoire courant et `!date` affiche l'heure

Soit le fichier `input` suivant:

```
# Test de la caluclatrice étendue
# un premier calcul:
x = 3 + 2
# et un second:
x * 100    # commentaire en fin de ligne
!(cd /tmp; pwd)
quit
# Le résultat du calcul suivant ne doit pas apparaître
x * x * x
```

L'éxécution de la commande `./calc < input` doit produire:

```
[1] Unexpected character x (120)
*** Error: syntax error
Bye
[2] Unexpected character x (120)
*** Error: syntax error
Bye
[3] /tmp
Bye
```

8 Espaces et langages de programmation

En général, dans les langages de programmation (excepté Python bien sûr :-) les caractères espaces et tabulations sont le plus souvent non significatifs, mais ce n'est pas toujours aussi clair que cela.

Soient les expressions C suivantes (où `x` et `y` sont toujours initialisées à 1 et 2 juste avant l'évaluation de l'expression.

Expression	Valeur	x	y	Valeur sans espace	x	y
x - 1						
x - -1						
x - - 1						
x - - - 1						
x - y						
x - -y						
x - - y						
x ---y						
-y						
- -y						
--y						
x -- - -- y						
x-- + --y						

Remplir la table précédente, et indiquez les endroits où la valeur calculée avec ou sans espaces diffèrent. Indiquez aussi, la valeurs de `x` et de `y` après l'évaluation de l'expression.

Remarque:

Toutes les expressions données ici sont valides, leur version sans espace sont peut être erroné.