

Théorie des langages — TP n°1

(<http://dpt-info.u-strasbg.fr/~alain/compilation/>)

Alain KETTERLIN (alain@dpt-info.u-strasbg.fr)

Premier semestre 2006-2007

Le but de ce TP est de se familiariser avec le générateur d'analyseur lexical Lex (ou flex dans sa version GNU), et de comprendre les fondements de l'analyse lexicale.

Présentation de lex

Lex est un générateur d'analyseur lexical : il prend en entrée une description lexicale (en résumé, un ensemble de couples expression-régulière/action) et produit une fonction d'analyse lexicale. Cette fonction est écrite en C et s'appelle `yylex()`. Elle peut ensuite être intégrée à n'importe quel programme.

Une description Lex a la forme suivante :

```
%{
... /* Déclarations C */
}%
... /* Déclarations Lex */
%%
... /* Règles lexicales */
%%
... /* Code C additionnel */
```

Les deux sections *Déclarations C* et *Code C additionnel* seront recopiées textuellement dans le fichier résultat. Les deux autres sections (*Déclarations Lex* et *Règles lexicales*) permettent de détailler le fonctionnement de l'analyseur lexical.

Règles lexicales

La troisième partie est la plus importante : elle contient la définition de l'analyseur lexical sous la forme d'un ensemble de couples contenant chacun une expression

régulière et une action.

```
exp-reg-1    action-1
exp-reg-2    action-2
...
```

Pour les expressions régulières, les constructions possibles sont résumées dans la table 1. Une action commence sur la même ligne que l'expression régulière à laquelle elle est associée, et se poursuit jusqu'à la fin de la ligne, ou jusqu'à l'accolade fermante correspondante si l'action commence par une accolade ouvrante. Le code C constituant l'action est reproduit textuellement par Lex dans le code C résultant, y compris s'il contient des erreurs (de syntaxe ou autres).

À partir d'une telle description, Lex produit un fichier C (appelé par défaut `lex.yy.c`), contenant une fonction `yylex()` qui ressemble à :

```
int yylex()
{
  while ( non fin de fichier )
  {
    trouver le plus long préfixe correspondant
      à une des expressions régulières
    si le lexème correspond à exp-reg-1 alors
      action-1
    sinon si le lexème correspond à exp-reg-2 alors
      action-2
    sinon ...
      ...
    sinon /* action par défaut (optionnelle) */
      afficher le premier caractère sur stdout
      éliminer les caractères utilisés de l'entrée
  }
  return 0;
}
```

Remarquez que l'action associée à un type de lexèmes peut contenir une instruction `return` (ce qui termine l'appel courant de `yylex()`). Si ce n'est pas le cas, `yylex()` continue à s'exécuter en recherchant le lexème suivant. Par convention, `yylex()` renvoie 0 lorsqu'elle atteint la fin du fichier, mais c'est au reste du programme de déterminer combien de fois elle doit être appelée.

Notez aussi que par défaut, Lex lit son entrée standard, et éventuellement écrit sur sa sortie standard.

Un seul caractère		
c	le caractère c	
\cdot	n'importe quel caractère	(sauf fin de ligne)
$[c_1c_2\dots c_k]$	c_1 ou $c_2 \dots$ ou c_k	$\equiv (c_1 c_2 \dots c_k)$
$[c_1-c_2]$	un caractère entre c_1 et c_2	
$[\wedge c_1c_2\dots c_k]$	ni c_1 ni $c_2 \dots$ ni c_k	
$[\wedge c_1-c_2]$	un caractère qui n'est pas entre c_1 et c_2	
$[:class:]$	avec $class$ dans $alpha$, $lower$, $upper$, $blank$, $space$, $digit$, $alnum$, $punct$, $graph$, $print$, $xdigit$, $cntrl$.	man $isalpha$ etc.
$\backslash 0$	le caractère de code 0	
$\backslash c$	si c est dans $nrtfvab$ alors le caractère correspondant en C, sinon le caractère c	$\backslash *$ représente le caractère $*$
$\backslash o_1o_2o_3$	code <i>octal</i> $o_1o_2o_3$	
$\backslash x_1x_2$	code <i>hexadécimal</i> x_1x_2	
$<<EOF>>$	la fin du fichier	
Une chaîne de caractères		
$\alpha\beta$	concaténation	
$(\alpha\beta)$	groupement	
$\alpha \beta$	alternative	
α^*	répétition (quelconque)	
α^+	répétition (non vide)	$\equiv \alpha\alpha^*$
$\alpha?$	option	$\equiv \alpha \epsilon$
$\alpha\{n,m\}$	répétition (entre n et m fois)	
$\alpha\{n\}$	répétition (exactement n fois)	$\equiv \alpha\{n,n\}$
$\alpha\{m,\}$	répétition (au moins m fois)	$\equiv \alpha\{m\}\alpha^*$
$\wedge\alpha$	α au début d'une ligne	
$\alpha\$$	α à la fin d'une ligne	
$"c_1c_2\dots c_k"$	la chaîne $c_1c_2\dots c_k$ littérale	$"a.b^*" \equiv a\backslash.b\backslash*$
$\{\lambda\}$	l'expression correspondant à la définition λ	

TAB. 1 – Syntaxe des expressions régulières Lex

Déclarations Lex

La section des *Déclarations Lex* contient deux types d'informations distincts.

Les options modifient le code produit. Il est rare d'avoir à en connaître le détail (ne serait-ce que pour des raisons de portabilité). Deux d'entre elles sont cependant utiles pour des raisons pratiques avec la version GNU de Lex (appelée *flex*). Elle s'écrivent :

```
%option nounput
%option noyywrap
```

Leur signification tient au fonctionnement interne de *flex* (voir la documentation). Sans ces options, les programmes contenant une fonction `yylex()` produite par *flex* doivent être liés à `libfl.a` (via l'option `-lfl`).

Les définitions quant à elles permettent de donner un nom à certaines parties d'expressions régulières. La syntaxe est immédiate. Voici un exemple :

```
DIGIT    [0-9]
%%
{DIGIT}+      return ENTIER;
{DIGIT}+"."{DIGIT}*  return FLOTTANT;
```

Notez que le nom défini doit être utilisé entouré d'accolades.

Accès au texte reconnu

Lorsque Lex déclenche une règle, cela signifie qu'il a trouvé une chaîne de caractères qui correspond à une des expressions régulières. Il exécute alors l'action associée. Pendant l'exécution de ce code (et seulement à ce moment là), le texte reconnu est placé dans la variable `yytext` (de type `const char *`). Le nombre de caractères de la chaîne est placé dans la variable `yylen`, de type `int`.

Invocation

Si votre fichier s'appelle `lexeur.lex`, et que vous lancez

```
flex lexeur.lex
```

vous obtenez un fichier `lex.yy.c` contenant la fonction `yylex()`. Je vous recommande un appel de la forme :

```
flex -olexeur.c -s -p -p -v lexeur.lex
```

(Voir la documentation pour la signification des options.)

1 Premier exemple : devenez spammeur

Écrivez un petit analyseur lexical qui permet d'afficher les adresses e-mail qui apparaissent dans un texte quelconque. Créez vous-même un fichier de test, ou utiliser une page web quelconque.

Vous développerez deux versions de votre programme. La première se contente d'afficher chaque adresse trouvée : le travail est fait dans l'action de la règle de `flex`. La seconde renvoie au programme principal chaque adresse e-mail trouvée : le programme est supposé les conserver dans une liste (ce qu'il est inutile de faire dans notre cas).

2 « Beautification » de code C

Le but de cet exercice est d'écrire un analyseur lexical capable de mettre en valeur certains éléments d'un programme C. Il s'agit de lire un fichier source C (sur l'entrée standard), et de produire en sortie du texte interprétable par un terminal ANSI (par exemple, votre `xterm`, `konsole` etc.). Vous trouverez sur la page web mentionnée en tête de ce document un fichier `ansitty.h` qui contient les séquences de caractères permettant de changer les caractéristiques du texte affiché. Votre programme doit reconnaître et afficher distinctement les mots-clés, les commentaires, les identificateurs, etc.

- Commencez par mettre en évidence les mots-clés (`while`, `for`, etc.) ainsi que les directives au pré-processeur (`#include` etc.) et testez votre programme, par exemple, sur le fichier C généré par `flex`. Que remarquez-vous ? (Cherchez par exemple les occurrences de `for`.)
- Modifiez votre analyseur lexical pour qu'il mette en évidence les chaînes de caractères constantes.
- Modifiez votre analyseur lexical pour qu'il mette aussi en évidence les commentaires, y compris lorsque ceux-ci s'étendent sur plusieurs lignes.
- Modifiez votre analyseur lexical pour qu'il affiche également, à la fin du traitement, une liste des fichiers inclus (par `#include` dans ses deux formes). Vous pouvez (exceptionnellement) utiliser un tableau de taille fixe pour la liste. Gérez séparément les fichiers de l'utilisateur et ceux du système.
- Enfin on veut également gérer les identificateurs, d'abord pour les mettre en valeur dans la sortie sur terminal, ensuite pour générer un index en fin de document. Un index contient la liste des identificateurs ainsi que la liste de leurs occurrences (identifiées par le numéro de ligne). Modifiez votre analyseur pour qu'il produise un index en fin de document. (Là encore, utilisez des structures de données aussi simples que possible.)

Enfin, réfléchissez à la question suivante : comment faudrait-il faire pour que l'on puisse distinguer, pour chaque identificateur, sa déclaration et ses utilisations ?

3 Mise en forme de texte

On veut écrire un petit analyseur lexical qui met en forme un texte, c'est-à-dire qu'il lit un texte en entrée, et écrit en sortie une transformation du texte. On adopte pour cela les règles suivantes :

- Les mots sont séparés par des blancs ou tabulations. Tout caractère non blanc fait partie d'un mot. Deux mots successifs peuvent être séparés par un nombre quelconque de blancs.
- Deux paragraphes sont séparés par une suite d'au moins deux caractères de fin de ligne.

Le programme doit produire en sortie un fichier dans lequel :

- Les mots sont séparés par un et un seul blanc.
- Les paragraphes sont séparés par exactement deux caractères de fin de ligne (c'est-à-dire exactement une ligne vide).
- Aucun mot n'est coupé, et aucune ligne ne dépasse 80 caractères (sauf si un seul mot est de longueur supérieure à 80 caractères, auquel cas il apparaît seul sur une ligne).

Le but ici n'est pas de reconnaître certains types d'unités lexicales, mais de traiter l'entrée standard et de réagir dans un certain nombre de cas : lorsqu'on a un mot, lorsqu'on lit des blancs séparant deux mots, et lorsqu'on lit des sauts de ligne séparant des paragraphes.

Voici un exemple de ce qui est attendu. Le texte est entrée est :

```
Bonjour Monsieur,

J'ai bien   reçu votre
courrier du 12 decembre, et j'en ai pris
    bonne note. Je reprends
contact avec vous aujourd'hui
pour...
```

Le résultat du programme est le suivant :

```
Bonjour Monsieur,

J'ai bien reçu votre [ ... etc ... ] bonne note. Je
reprends contact avec vous aujourd'hui pour...
```

Note : dans cet exercice, c'est la fonction `yylex()` qui réalise la totalité du travail. Écrivez une fonction `main()` qui appelle (une seule fois) `yylex()`.