Théorie des langages
TP 1 - Utiliser `flex`

Jonathan Fabrizio et Adrien Pommellet, EPITA

9 novembre 2022

Pensez à installer `flex` au préalable avec votre gestionnaire de paquets. Si cette bibliothèque ne s'avère pas disponible sur votre poste, utilisez l'instruction suivante :

```
nix-shell -p flex
```

1 À propos de `flex`

`flex` est un outil qui permet d'engendrer automatiquement des analyseurs lexicaux, aussi appelés *lexers*. Ces programmes sont utilisés pour détecter des motifs dans un flux d'entrée et appliquer des actions dépendant des motifs identifiés. `flex` manipule des fichiers `.l` de la forme suivante :

```
flex options
%{
  C declarations
%}
flex declarations
%%
%{
  C prologue
%}
flex rules
%%
custom C code
```

Le concept de *règle* est essentiel au fonctionnement de `flex`. Une règle associe à un motif exprimé par une *expression régulière* une suite d'instructions écrites en `C` entre `{}`. `flex` utilise la syntaxe habituelle pour les expressions régulières (voir à ce sujet le manuel). Par exemple, ces règles permettent de détecter des mots écrits respectivement en minuscules et en majuscules latines, puis d'afficher un message pertinent :

```
[a-z]+ {printf("Word found.\n"); printf("Lower case.\n");}
[A-Z]+ {printf("Word found.\n"); printf("Upper case.\n");}
```

`flex` permet de produire un fichier `C` que l'on peut ensuite inclure dans un projet ou exécuter seul selon les options choisies. Pour créer le fichier `output.c` à partir du fichier `input.l`, on utilise la commande suivante :

```
flex [options] -o output.c input.l
```

Il ne reste plus qu'à compiler `output.c` (inutile d'utiliser trop d'options) et exécuter le programme. Il est fortement recommandé d'écrire un `makefile` dédié pour automatiser ce protocole.

2 Premier contact

Commencez par copier le code suivant dans un fichier `example1.l` :

```
%option nounput noinput noyywrap

%{
  #include <stdio.h>
%}

LETTER_B_OR_C [BC]

%%

%{
  printf("Looking for a pattern...\n");
%}

A {printf("A found.\n");}
{LETTER_B_OR_C} {printf("B or C found.\n");}
D+ {printf("One or more D found.\n");}

%%

int main() {
  while (yylex());
  return 0;
}
```

Les options `nounput`, `noinput`, et `noyywrap` permettent respectivement de désactiver l'insertion manuelle de nouveaux caractères dans le flux d'entrée, d'empêcher l'utilisateur de contourner l'analyse lexicale, et d'imposer l'arrêt du lexer à la lecture du premier symbole de fin de fichier. `LETTER_B_OR_C [BC]` est une abréviation qui permet d'utiliser `{LETTER_B_OR_C}` dans les règles pour faire appel à l'expression régulière `[BC]`. La fonction `yylex()` lit le flux d'entrée par des appels itérés et traite une règle par appel.

Cet analyseur lexical utilise le flux d'entrée par défaut `stdin` et peut être interrompu avec `CTRL+D` (auquel cas `yylex()` renvoie zéro).

Question 1. Appliquez `flex` au fichier `example1.l`, puis compilez et exécutez le programme ainsi obtenu. Testez-le ensuite sur les entrées `ACABDDD`, `123`, et `A123C`. Quelle conclusion pouvez-vous en tirer sur le fonctionnement du lexer ? Quelle est l'action par défaut effectuée sur le flux d'entrée si aucune règle ne s'applique ?

3 Votre premier analyseur

Commençons par créer un analyseur qui détecte des motifs simples.

3.1 Structure des règles

Partez du squelette de fichier suivant, auquel vous devrez ajouter des règles entre les `%%` :

```
%option nounput noinput noyywrap

%{
  #include <stdio.h>
%}

%%

/* Replace this comment with actual flex rules. */

%%

int main() {
  while (yylex());
  return 0;
}
```

Question 2. Ajoutez des règles spécifiques pour les motifs `2` et `4`. Puis compilez et testez votre lexer sur l'entrée `1234526447824`.

Question 3. Ajoutez une règle spécifique pour le motif `42`. Puis compilez et testez votre lexer sur l'entrée `42`. Comment expliquer le comportement du lexer ?

Question 4. Ajoutez une règle propre aux entiers non signés. Puis compilez et testez votre lexer sur les entrées `42` et `4242`. Comment expliquer le comportement du lexer ? Que se passe t-il si la règle pour les entiers est écrite avant celle propre à `42` ? Et après ?

Question 5. Modifiez la règle précédente de manière à afficher l'entier qui vient d'être lu. Gardez en tête que le dernier motif lu par le lexer est une chaîne de caractères accessible grâce à la variable `yytext`.

3.2 Étendre les règles

Vous avez sans doute remarqué qu'une action par défaut est effectuée à chaque fois qu'une entrée ne peut être associée à un jeton connu.

Question 6. Remplacez cette action par une règle que vous avez conçue (en utilisant l'une des expressions régulières décrites dans le manuel). Vérifiez que la sortie par défaut est désormais inactive.

Question 7. Enrichissez votre analyseur lexical en ajoutant des règles pour gérer sur le flux d'entrée les symboles `+` (écrit `"+"` dans les expressions régulières afin de ne pas le confondre avec l'opérateur de Kleene `+`), `-` (écrit `"-"` dans les expressions régulières afin de ne pas le confondre avec l'opérateur `-` sur les intervalles) et les suites d'espaces blancs. Puis compilez et testez votre lexer.

4 Un décompte des mots

L'objectif de cette partie est l'écriture d'un analyseur lexical capable de compter le nombre de mots (écrits dans l'alphabet latin) dans une phrase d'entrée. Il vous faudra compléter le fichier suivant :

```
%option nounput noinput noyywrap

%{
  #include <stdio.h>
%}

%%

%{
  printf("Please enter a sentence.\n");

  /* Initialise a counter here */
%}

/* Replace this comment with actual flex rules. */

%%

int main() {
  while (yylex());
  return 0;
}
```

Il vous faudra initialiser un compteur dans le prologue `C`, puis l'incrémenter en fonction des motifs lus. Attendez le symbole `\n` en fin de ligne pour afficher le nombre de mots, puis n'oubliez pas de réinitialiser le compteur.

Question 8. Programmez un lexer qui affiche le nombre de mots dans une ligne d'entrée composée uniquement de lettres de l'alphabet latin et d'espaces (tout autre caractère provoque une erreur et un arrêt du lexer par l'instruction `return 0`). Compilez-le, exécutez-le, et testez-le.

5 Une calculatrice primitive

Nous souhaitons enfin programmer une calculatrice rudimentaire capable d'effectuer des additions et des soustractions sur les entiers naturels. Toute expression mal formée ou symbole autre qu'une soustraction, une addition, ou une suite de chiffres doit provoquer une erreur. On cherche à obtenir une sortie de la forme :

```
12+36-55
= -7
```

Il vous faudra compléter le fichier suivant :

```
%option nounput noinput noyywrap

%{
  #include <stdio.h>

  /* Fill in the blanks */
%}

%%

%{
  printf("Please enter a simple arithmetic expression.\n");

  /* Fill in the blanks */
%}

/* Replace this comment with actual flex rules. */

%%

int main() {
  while (yylex());
  return 0;
}
```

Notons que le comportement du lexer va dépendre non seulement du dernier motif lu, mais aussi du motif vu précédemment : par exemple, tout opérateur doit être suivi d'un entier. De plus, l'entrée doit toujours commencer par un entier, et le résultat du calcul doit être affiché une fois la ligne entièrement lue.

Vous aurez besoin de deux variables globales : une pour gérer l'état du lexer (parmi une liste d'états possibles décrites dans un `enum` en début du fichier), et une autre pour conserver le résultat des calculs courants. La fonction `C atoi` vous sera aussi utile.

Question 9. Codez un lexer qui implémente la calculatrice décrite précédemment. Compilez-le, exécutez-le, et testez-le.