Théorie des langages
TP 1 - Utiliser *flex*

Jonathan Fabrizio et Adrien Pommellet, EPITA

3 novembre 2022

Pensez à installez *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 Une introduction à 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 passer des *jetons* au *parser* au lieu d'une entrée brute. Flex manipule des fichiers `.l` de la forme suivante :

```
Options
%{
  C declarations
%}
Flex declarations
%%
%{
  C prologue
%}
Regular expressions and actions
%%
Custom C code
```

Flex utilise la syntaxe habituelle pour les expressions régulières (voir à ce sujet le manuel). À chaque règle, on associe une action écrite en C entre `{}`. Par exemple, les règles suivantes permettent de détecter des mots écrits respectivement en minuscules et en majuscules latines, puis d'afficher un message pertinent :

```
%%
[a-z]+ {printf("Lower case");}
[A-Z]+ {printf("Upper case");}
%%
```

Le motif détecté par le lexer est stocké dans la variable `yytext`. Pour lancer flex, 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 Votre premier programme flex

Nous allons écrire un premier interpréteur lexical avec flex. Il va falloir compléter le fichier `.l` suivant :

```
%option nounput noinput noyywrap

%{
  #include <stdio.h>
```

```
%}

BLANK [ \t]

%%

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

/* Fill in the blanks */

%%

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. `BLANK [ \t]` est une abréviation optionnelle pour les espaces et les tabulations à laquelle on peut faire appel dans une expression régulière avec l'instruction `{BLANK}`. La fonction `yylex()` lit le flux d'entrée par des appels itératifs et traite un jeton par itération.

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. Copiez le code du lexer, compilez-le, exécutez-le et testez-le. Quelle est l'action par défaut effectuée sur le flux d'entrée ?

2.1 Un premier essai

Commençons par analyser des motifs simples.

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 à l'entier `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 non 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 accessibles grâce à la variable `yytext`.

2.2 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. Vérifiez que la sortie par défaut est désormais inactive.

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

3 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 */
%}

/* Fill in the blanks */

%%

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). Compilez-le, exécutez-le, et testez-le.

4 Une calculatrice primitive

Nous souhaitons enfin programmer une calculatoire 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 */
%}
```

```
/* Fill in the blanks */

%%

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, un entier doit toujours être suivi d'un symbole `+` ou d'un `-`, et réciproquement, 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.