Théorie des langages
TP 4 - Combiner *flex* et *bison*

Jonathan Fabrizio et Adrien Pommellet, EPITA

17 décembre 2021

Pensez à installez *flex*, *bison* et *graphviz* au préalable avec votre gestionnaire de paquets. Si ces bibliothèques ne s'avèrent pas disponibles sur votre poste, utilisez l'instruction suivante :

```
nix-shell -p flex graphviz bison
```

1 Interfacer le lexer et le parser

L'objectif de ce TP est d'interfacer un *lexer* engendré par flex avec un *parser* engendré par bison de manière à implémenter une calculatrice basique. Le lexer lit un flux d'entrée et produit un flux de jetons que le parser va ensuite interpréter pour tester son appartenance à une grammaire donnée.

1.1 Définition de la grammaire

On considère la grammaire suivante :

$$
\begin{aligned}
E &\to E + E && (1)\\
&\mid E * E && (2)\\
&\mid E - E && (3)\\
&\mid E / E && (4)\\
&\mid \mathrm{n} && (5)
\end{aligned}
$$

Elle engendre les expressions arithmétiques sur les entiers. Notez que de manière similaire à flex, il est possible dans un fichier bison d'associer des actions aux règles de grammaire, qui seront exécutées lors de leur réduction.

```
A:
a {printf("Letter a spotted.");}
```

Il ne faut bien sûr pas oublier d'inclure les bibliothèque nécessaires à l'exécution du code C en ajoutant dans le prologue du fichier `.y` le code suivant :

```
%code top {
  #include <stdio.h>
}
```

Il est encore une fois recommandé d'utiliser un fichier `Makefile` personnalisé de manière à automatiser le processus de compilation.

Question 1. Écrivez un fichier `.y` pour la grammaire suivante et compilez-le avec bison en garantissant qu'il n'y a pas le moindre conflit.

Question 2. Ajoutez à chaque règle du fichier `.y` une action `printf` qui la caractérise, puis compilez-le avec bison. Ne cherchez pas à compiler le fichier `.c` produit tant qu'on ne vous le demande pas !

1.2 Définition des jetons et du lexer

Il nous faut désormais définir des *jetons*. On souhaite en effet donner au parser des jetons pré-interprétés et non un flux d'entrée brut : à chaque fois que le lexer va reconnaître un certain motif régulier dans le flux d'entrée, il va envoyer au parser un jeton qui décrit ce motif. Par exemple, si l'on ajoute le prologue suivant au fichier bison :

```
%define api.token.prefix {TOK_}

%token INTEGER "n"
```

Cet ajout va permettre d'interagir avec la règle flex suivante :

```
[0-9]+ {return TOK_INTEGER;}
```

Le jeton `TOK_INTEGER` associé à l'expression régulière `[0-9]+` va être donné au parser qui l'interprétera comme étant le symbole terminal `n` utilisé dans ses règles de grammaire. Ces jetons, bien qu'étant produits par le lexer, sont toutefois définis au niveau du parser. Il faut donc que bison crée un fichier d'en-tête `.h`. Pour ce faire, on insère dans le prologue du fichier `.y` les options suivantes :

```
%defines
%define api.pure full
```

Puis, une fois le fichier `header.h` produit, on inclut dans l'en-tête du fichier `.l` associé au lexer les instructions suivantes :

```
%option noyywrap nounput noinput

%{
  #include "header.h"
%}
```

N'hésitez pas à revoir le premier TP 1 ou à lire les manuels de flex et de bison pour mieux comprendre le sens des différentes options utilisées.

Question 3. Définissez dans le prologue du fichier `.y` des jetons associés à tous les terminaux utilisés par la grammaire. Puis compilez-le avec bison et assurez-vous qu'un fichier `.h` a bien été généré.

Question 4. Écrivez un lexer au format `.l` qui reconnaît les entiers naturels ainsi que les symboles `+`, `-`, `*`, et `/`, et renvoie les jetons associés à ces symboles.

1.3 Intégration du lexer

Pour que le parser puisse directement s'interfacer avec le lexer, il faut déclarer dans le fichier bison la fonction principale `yylex` du lexer ainsi qu'une fonction d'erreur `yyerror`. On ajoute donc au prologue du fichier `.y` les instructions suivantes :

```
%code provides {
  #define YY_DECL enum yytokentype yylex()
  YY_DECL;
  int yyerror(const char * s);
}
```

où `enum yytokentype` est une énumération engendrée automatiquement des différents jetons décrits dans le fichier `.y` file. La fonction `int yyerror(const char * s)` écrite manuellement doit afficher le message d'erreur `s` et renvoie `0`. Il ne reste plus qu'à faire appel dans la fonction principale `main` à la fonction `int yyparse()` pour lancer le parsing : cette dernière renvoie alors `0` si le parsing s'est déroulé correctement.

Question 5. Écrivez la fonction `int yyerror(const char * s)` dans l'épilogue du fichier `.y`, après les règles de grammaire.

Question 6. Ajoutez une fonction `main` dans l'épilogue du fichier `.y`, après les règles de grammaire. Compilez (avec bison, puis flex, puis un compilateur C) et testez le programme ainsi produit sur différents exemples.

2 Adaptation de la grammaire

Notons que le flux d'entrée passé au parser est constitué de plusieurs lignes, chacune se finissant par le symbole `\n` ; il faut donc modifier la grammaire originale pour gérer cette particularité, sans quoi une erreur de syntaxe apparaîtra dès la lecture de la seconde ligne. On considère donc la grammaire :

$$
\begin{aligned}
\mathit{Lines} &\to \mathit{Lines}\ L && (1)\\
&\mid \varepsilon && (2)\\
L &\to E\ \text{eol} && (3)\\
E &\to E + E && (4)\\
&\mid E * E && (5)\\
&\mid E - E && (6)\\
&\mid E/E && (7)\\
&\mid \text{n} && (8)
\end{aligned}
$$

Il se peut que vous ayez à définir un nouveau jeton pour le terminal `eol`. Le symbole ε est représentable par la commande `%empty`.

Question 7. Modifiez les fichier `.y` et `.l` de manière à pouvoir parser cette grammaire. Compilez (avec bison, puis flex, puis un compilateur C) et testez le programme ainsi produit sur différents exemples.

3 Une implémentation d'une calculatrice

Notre programme peut reconnaître des motifs engendrés par la grammaire sur le flux d'entrée, mais il ne peut pas encore effectuer d'opérations sur les entiers.

3.1 Manipulation des entités lexicales

Certains jetons renvoyés par le lexer devraient être capables de *porter* des valeurs entières que l'on peut ensuite passer au parser. Pour ce faire, nous allons définir un type C d'`union` que nous allons associer au jeton lié aux entiers. Nous allons écrire dans le prologue du fichier `.y` file, avant de définir les jetons :

```
%union {
  int ival;
}
```

Ce type sera alors automatiquement exporté vers l'en-tête `.h`. Nous pouvons alors autoriser un jeton `X` à porter une valeur entière en modifiant sa définition :

```
%token<ival> X
```

Notons qu'il est aussi possible d'autoriser un symbole non-terminal `Y` à porter une valeur entière en écrivant l'instruction suivante après les définitions des jetons :

```
%type<ival> Y
```

Le lexer peut accéder à la composante `x` de l'union portée par les jetons grâce à l'instruction `yyval->x`. Enfin, il faut penser à modifier la déclaration de la fonction `yylex` dans le prologue `%code provides` par `#define YY_DECL enum yytokentype yylex(YYSTYPE *yylval)}` car elle doit désormais garder en mémoire les entiers lus.

Question 8. Définissez et associez la valeur entière `ival` au terminal `n` et aux non-terminaux `E` et `L` dans le fichier `.y`.

Question 9. Modifiez le fichier `.l` de manière à associer le motif `yytext` lu à l'entier `ival` dès qu'un jeton entier est identifié. Vous pouvez utiliser pour ce faire la fonction `atoi` de la bibliothèque `stdlib`, en prenant bien soin de l'inclure dans le fichier `.l`.

3.2 Exécution des calculs

À chaque fois que le parser applique une règle de grammaire pour réduire une expression, il est possible d'effectuer certaines actions sur les valeurs portées par les jetons et les non-terminaux associés. Par exemple :

```
IncrementX:
  X {$$ = $1 + 1;}
```

`$$` représente la valeur portée par la partie gauche `IncrementX`, `$1` la valeur portée par le premier symbole de la partie droite, c'est-à-dire `X`.

Question 10. Écrivez dans le fichier `.y` les actions appropriées à associer aux règles de grammaire (4) à (8). Méfiez-vous de la division par 0 : utilisez la suite d'instructions `yyerror("Some message."); YYERROR;` pour renvoyer une erreur.

Question 11. Modifiez le fichier `.y` de manière à afficher le résultat du calcul courant à chaque fin de ligne. Puis compilez et testez le programme.

Question 12. Modifiez la grammaire, puis le lexer et le parser, de manière à gérer les parenthèses dans les expressions arithmétiques. Puis compilez et testez le programme. Enfin fini !