Théorie des langages
TP 2 - Une introduction à *bison*

Jonathan Fabrizio et Adrien Pommellet, EPITA

25 octobre 2021

Pensez à installez *bison* et *graphviz* auparavant avec votre gestionnaire de paquets.

1 Une présentation de bison

Bison est un outil qui génère automatiquement des *parsers* LR à partir de grammaires décrites dans des fichiers `.y` de la forme :

```
Optional prologue: options, declarations, priority rules...
%%
Rules
%%
Optional epilogue: custom code...
```

Les terminaux sont écrits entre guillemets. Le symbole `:` remplace la flèche $\rightarrow$ dans la notation BNF, mais le sens de | reste le même. Un point virgule `;` doit conclure les suites de règles de la forme $X \rightarrow a \mid b \mid \dots$ associées à un même non-terminal. Par exemple, si l'on considère la grammaire suivante :

$$S \rightarrow (S) \tag{1}$$
$$\mid n \tag{2}$$

Une représentation sous forme de fichier `.y` est alors :

```
%%
S:
   "(" S ")"
  | "n"
;
%%
```

Notez que bison ajoute toujours implicitement une règle :

```
$accept: S $end
```

Qui représente la règle habituelle $Z \rightarrow S\$$ où S est le premier non-terminal observé. Pour créer un parser avec bison, on utilise la commande suivante :

```
bison [options] input.y -o output.c
```

Bison va alors produire un parser `output.c` pour la grammaire `input.y`. Si des conflits sont repérés, l'option `-Wcounterexamples` peut être utilisée pour afficher des contre-exemples. L'option `--verbose` permet de produire un fichier texte décrivant l'automate LR associé. `--graph` engendre une représentation graphique au format `.dot` du fichier. Il est possible de la convertir au format `.png` en utilisant la commande `dot` du paquet graphviz.

```
dot -Tpng output.dot > output.png
```

Il est hautement recommandé de créer un fichier `makefile` dédié pour automatiser le processus de compilation.

Question 1. Calculez l'automate `LR(0)` associé à la grammaire décrite précédemment, puis écrivez le fichier `.y` correspondant, compilez-le, et vérifiez votre réponse grâce à la sortie graphique.

2 Priorité des opérations associatives

On peut tolérer n conflits shift-reduce en insérant l'option suivante dans le prologue :

```
%expect n
```

Mais on préfère souvent garantir une absence totale de conflits :

```
%expect 0
```

Question 2. Écrivez un fichier `.y` associé à la grammaire suivante. Essayez de le compiler. Y a t-il un conflit ? Si oui, pourquoi ? Essayez de trouver des contre-exemples en utilisant l'option appropriée.

$$\begin{aligned} E &\to E + E && (1)\\ &\mid E * E && (2)\\ &\mid \text{expression} && (3) \end{aligned}$$

L'une des manières les plus courantes de résoudre des conflits est l'utilisation de règles d'associativité et de priorité. Considérons le prologue suivant :

```
%right "a" "b"
```

Cette option précise que les symboles `a` et `b` sont associatifs à droite : tout conflit shift-reduce impliquant la lecture d'un symbole `a` ou `b` et une règle dont le dernier symbole terminal est `a` ou `b` sera arbitré en faveur du shift. L'option `left` permet de définir l'associativité à gauche et donnera plutôt la priorité à la réduction. La priorité dépend également de l'ordre dans lequel les opérateurs sont déclarés :

```
%left "a"
%left "b"
```

Cette option précise que `a` sont tous les deux `b` associatifs à gauche, et que `b` est prioritaire sur `a`.

Question 3. Résolvez les conflits dans le fichier `.y` précédent, et garantissez leur absence. Puis compilez le fichier et affichez l'automate produit.

3 Règles de précédence

Question 4. Écrivez un fichier `.y` associé à la grammaire suivante. Essayez de le compiler. Y a t-il un conflit ? Si oui, pourquoi ? Essayez de trouver des contre-exemples en utilisant l'option appropriée.

$$\begin{aligned} E &\to \text{if } E \text{ then } E && (1)\\ &\mid \text{if } E \text{ then } E \text{ else } E && (2)\\ &\mid \text{expression} && (3) \end{aligned}$$

Il arrive que des opérateurs ne soient pas associatifs (quel sens donner à `E then E then E` ?) mais que l'on ait malgré tout besoin de définir une forme de précédence entre eux. L'option `nonassoc` permet de renvoyer une erreur à l'exécution si un conflit d'associativité impliquant les opérateurs décrits a lieu :

```
%nonassoc "a"
```

L'option `precedence` crée plutôt des erreurs à la compilation ; un opérateur peut en effet être impliqué dans des conflits d'associativité à l'insu de l'auteur de la grammaire :

```
%precedence "a"
```

Les options `nonassoc` et `precedence` peuvent toutes les deux être utilisés pour définir des priorités sur des opérateurs qui ne sont pas censés être associatifs :

```
%precedence "a"
%precedence "b"
```

Question 5. Sans utiliser les options `%left` et `%right`, résolvez les conflits dans le fichier `.y` précédent et garantissez leur absence. Puis compilez le fichier et affichez l'automate produit.

Question 6. Est-il également possible de modifier la grammaire et le langage associé de manière à prévenir de tels conflits sans utiliser de règles de précédence ? Pensez aux solutions utilisés par divers langages de programmation.

Question 7. Écrivez un fichier `.y` associé à la grammaire suivante en garantissant l'absence de conflits. Puis compilez-le et affichez l'automate produit.

$$E \to E \,?\, E : E \quad (1)$$
$$\mid \text{expression} \quad (2)$$

4 Un parser LR(1)

Question 8. Écrivez un fichier `.y` associé à la grammaire suivante. Essayez de le compiler. Y a t-il un conflit ? Si oui, pourquoi ? Essayez de trouver des contre-exemples en utilisant l'option appropriée.

$$S \to \text{proc } A \,. \quad (1)$$
$$\mid \text{proc } P \,; \quad (2)$$
$$\mid \text{macro } A \,; \quad (3)$$
$$\mid \text{macro } P \,. \quad (4)$$
$$A \to \text{id} \quad (5)$$
$$P \to \text{id} \quad (6)$$

Remarquez que bison calcule un automate LALR(1) par défaut. On peut plutôt appliquer un algorithme LR(1) en insérant l'option suivante dans le prologue :

```
%define lr.type canonical-lr
```

Question 9. Résolvez les conflits dans le fichier `.y` précédent et garantissez leur absence. Puis compilez le fichier et affichez l'automate produit.

Question 10. Écrivez un fichier `.y` associé à la grammaire suivante. Essayez de le compiler. Y a t-il un conflit ? Si oui, pourquoi ? Est-il possible de trouver des contre-exemples ? Pourquoi ?

$$E \to T \,[\, E \,] \text{ of } E \quad (1)$$
$$\mid L \quad (2)$$
$$L \to \text{id} \quad (3)$$
$$\mid L \,[\, E \,] \quad (4)$$
$$T \to \text{id} \quad (5)$$

Question 11. Réécrivez la grammaire précédente de manière à accepter le même langage tout en ayant un fichier `.y` qui puisse être compilé, puis affichez l'automate produit.