TD4 : Parsing

2023-2024

Syntaxe

La quasi-totalité de ce TD est à réaliser dans le dossier `td_4`, à l'exception (évidemment) du parseur du langage du cours qui est lui à réaliser dans `compiler/course_language/Parser.mly`. La difficulté est à peu près progressive. Néanmoins, si vous voulez vous attaquer à des parties du parseur de programmes, vous pouvez le faire en parallèle du reste du TD.

D'une manière générale, tous les fichiers d'explication du parseur générés par menhir se trouveront dans le dossier `parser_files`.

Exercice 1: Mise en jambes : $a^n b^n$

On considère le langage

$$L_1 = \{w = a^n b^n \mid n \in \mathbb{N}, \}$$

1. Pourquoi est-il impossible de réaliser un lexer (avec `OCamllex`) acceptant uniquement les mots de L_1 ?

2. Il n'est pas courant d'avoir à utiliser la grammaire G_1. Pourquoi ce résultat nous intéresse-t-il tant?

3. Écrire une grammaire algébrique G_1 qui produit le langage L_1. Astuce: vous savez qu'une grammaire algébrique est un tuple (V, Σ, S, R) où V est un ensemble de symboles de réécriture. Il existe deux variantes de cette grammaire, une avec ε-production, une autre sans.

   Cette grammaire est implémentée dans le répertoire `td_4/first_parser`.

4. Ouvrez `Parser.mly` et lisez-le. Observez que l'écriture de la grammaire est assez naturelle.

5. Compilez avec la commande `make parser_visualiser.out`. Cela produira un exécutable `parser_visualiser.out` avec lequel vous pourrez visualiser l'analyse d'un mot de L_1, ainsi que des fichiers décrivant l'automate généré, ainsi que ses potentiels conflits.

   (a) Pour utiliser l'exécutable, lancez `./parser_visualiser.out` pour avoir un message d'usage. Cet exécutable sera commun à toutes les grammaires de ce TD, et permet de choisir la grammaire à utiliser, ainsi que la lecture directement depuis un argument ou dans un fichier.

   Visualisez grâce à cet outil l'analyse de plusieurs mots, soit qui appartiennent à L_1, soit qui n'y appartiennent pas. Les 'a' et 'b' peuvent être remplacés respectivement par '(' et ')'.

(b) Le fichier `parser_files/first_parser.automaton`. Il contient l'ensemble des états ainsi de les transitions par des tokens ou par des symboles de la grammaire. Comparez avec l'automate décrit dans le polycopié du cours (c'est le même si on ignore pour le moment les éléments entre [ ]).

(c) Pour visualiser le même automate shift-reduce sous forme graphique, ouvrez le fichier `parser_files/first_parser.dot` avec le programme `xdot` ou `graphviz` s'ils sont installés; sinon produisez le fichier `simple_parser.jpg` avec la commande suivante:
`dot -T jpg -o simple_parser.jpg simple_parser.dot`

(d) Le fichier `parser_files/first_parser.conflicts` contient une description des conflits présents dans l'automate (LR1) généré s'il y en a. Ici il n'y en a pas (la grammaire fournie est LR0), le fichier est donc vide.

6. Changez maintenant la grammaire décrite dans `Parser.mly` pour correspondre à la grammaire plus standard avec une ε-production. Observez la différence entre l'automate et les arbres produits avec la version précédente.

Exercice 2: Bac à sable

Le dossier `td_4/custom_parser` contient la même grammaire que celle du premier exercice. Cette grammaire s'utilise dans `parser_visualiser.out` (comme tout ce sujet). Ce dossier est là pour vous permettre de tester diverses grammaires sans changer ce qui vous a été fourni. N'hésitez pas à la modifier à votre guise et à l'utiliser pour tester diverses grammaires (vous devrez également adapter le lexer). Elle produira les mêmes choses que le reste du sujet.

Exercice 3: Langage de Dyck

Un langage de Dyck L_D est un langage sur l'alphabet $\Sigma = \{(,)\}$ tel que

- $\epsilon \in L_D$
- Si u et $v \in L_D$, alors (u) $v \in L_D$

Bref, le langage de Dyck contient toutes les chaînes contenant uniquement des parenthèses, et tel que ces parenthèses soient bien placées:

- Il y a autant de parenthèses ouvrantes que fermantes.
- Tout préfixe contient plus de parenthèses ouvrantes que de fermantes.

exemple: `(()(((())))` en est, `())` n'en est pas.
Comme dans l'exercice précédent, '(' peut être remplacé par 'a' et ')' par 'b'.

Cet exercice est à réaliser avec l'exécutable `parser_visualiser.out`, et les fichiers des grammaires sont dans le dossier `td_4/dyck_parser`. Ces fichiers sont déjà fournis, il s'agit d'un exercice d'observation.

Il y a 4 fichiers de parseurs : `Tokens.mly`, qui contient la définition des tokens, qui sont communs aux trois grammaires, et permet d'utiliser le même lexer pour les trois; et `Dyck_simple.mly`, `Dyck_left` et `Dyck_right` qui sont trois grammaires différentes acceptant le langage de Dyck.

1. Comparez les arbres produits sur un même mot par les trois grammaires fournies. Vous avez des fichiers d'exemples fournis dans `dyck_files`, mais vous pouvez tester avec vos propres mots.

2. Vous avez pu observer que la compilation mentionne des conflits résolus arbitrairement. C'est la grammaire `Dyck_simple.mly` qui les produit. Regardez le fichier `Dyck_simple.conflicts` qui les explique, ainsi que les fichiers `*.automaton` qui décrivent les automates LR1 et mentionnent explicitement ces conflits (ici, seul l'état 6 est concerné). Essayez de comprendre comment fonctionnent ces automates. Les fichiers `*.dot` correspondants vous permettent si besoin d'en visualiser l'automate (cf exercice précédent).

3. Pourquoi les autres grammaires qui implémentent les grammaires de Dyck ne posent pas ce problème de conflit ? En particulier, observez que la grammaire `Dyck_right` correspond à celle décrite dans le cours et ne contient pas ici de conflit (puisque Menhir ne se limite pas à LR0).

4. Comment le conflit de `Dyck_simple` est-il résolu par Menhir ? Est-ce satisfaisant ?

Exercice 4: Expressions, termes, facteurs

On souhaite maintenant écrire une grammaire pour les expressions arithmétiques de notre langage. On va dans un premier temps se restreindre aux nombres entiers. Les expressions sont l'ensemble des formules contenant les nombres, les opérateurs binaires $+, -, \times, /$, les opérateurs unaires $-$ et $^{-1}$ (noté `^-1`) ainsi que le parenthèsage quand celui-ci est nécessaire.

Plus précisément, l'ensemble des expression e est le plus petit ensemble tel que

- si e_1 et e_2 sont deux expressions, alors $e_1 + e_2$ est une expression
- si e_1 et e_2 sont deux expressions, alors $e_1 - e_2$ est une expression
- si e est une expression, alors $-e$ est une expression
- si e_1 et e_2 sont deux expressions, alors $e_1 \times e_2$ est une expression
- si e_1 et e_2 sont deux expressions, alors e_1/e_2 est une expression
- si e est une expression, alors e`^-1` est une expression
- si e est une expression, alors (e) est une expression
- tout nombre est une expression

L'usage des opérateurs est tel que la multiplication est toujours prioritaire sur l'addition et que les deux opérateurs sont associatifs à gauche. Par exemple $2 + 3 \times 4 = 2 + (3 \times 4)$, $2 \times 3 + 4 = (2 \times 3) + 4$ et $2 - 3 - 4 = (2 - 3) - 4$.

Cet exercice est à réaliser dans le dossier `td_4/etf_parser`, sauf le premier exercice qui peut être réalisé dans `td_4/eee_parser`.

1. Dans un premier temps, donnez une grammaire naïve générant les expressions (avec un seul non-terminal; en plus de main) correspondant à la description ci-dessus. Observez que cette grammaire génère des conflits et que ceux-ci ne sont pas résolus de manière satisfaisante (en particuliers, sur les exemples ci-dessus). Vous pouvez vous limiter dans cet exercice à uniquement l'addition et la multiplication (pour obtenir des automates plus lisibles). Ne jetez pas cette grammaire, mettez-la dans `eee_parser`, nous en aurons besoin plus tard.

Dessinez l'automate LR0 correspondant à cette grammaire (avec seulement + et ×). Où sont les conflits ? Ces conflits peuvent ils être résolus avec la connaissance de la prochaine lettre à lire (indice : Menhir n'y arrive pas non plus).

Le reste de l'exercice va consister à explorer une grammaire qui résout ce problème.

2. Écrire la grammaire G_2 en utilisant la définition suivante des expressions arithmétiques qui utilise trois ensembles: les expressions e, les termes t et les facteurs f:

   L'ensemble des expressions arithmétiques e est le plus petit ensemble tel que

   - si `t` est un terme, et `e` une expression, alors `e + t` est une expression.
   - si `t` est un terme, alors `- t` est une expression.
   - si `t` est un terme, alors `t` est une expression.

   L'ensemble des termes t est le plus petit ensemble tel que

   - si `f` est un facteur, et `t` un terme, alors `t` × `f` est un terme.
   - si `f` est un facteur, et `t` un terme, alors `t / f` est un terme.
   - si `t` est un terme, alors `t^-1` est un terme.
   - si `f` est un facteur, alors `f` est un terme.

   L'ensemble des facteurs f est le plus petit ensemble tel que

   - si `e` est une expression, alors `(e)` est un facteur.
   - tout nombre est un facteur.

3. Implémentez cette grammaire dans `etf_parser/Parser.mly`, en ne prenant pour le moment que les additions et les multiplications. Observez qu'il n'y a maintenant plus de conflits. En vous servant du fichier `.automaton` produit, dessinez l'automate produit. Cet automate est-il LR0 ?

4. Terminez l'implémentation de toutes les expressions avec la grammaire ETF. Observez qu'il n'y a pas de conflits et que les expressions sont maintenant analysées correctement.

5. Sans faire l'exercice, imaginez maintenant que nous ne manipulons pas seulement des expressions arithmétiques, mais des expressions de comparaison entre les nombres, et des expressions booléennes. Il est évident que l'exercice va devenir long et pénible et qu'une forme de redondance inutile va s'installer.

   En effet, par ces trois définitions des expressions arithmétiques, nous avons défini en fait deux choses:

   (a) La compositionalité des expressions complexes, c'est-à-dire le fait que des expressions sont composées d'autres expressions.
   (b) La priorité des opérateurs, c'est-à-dire le fait que l'on calcule les facteurs avant les termes, eux-mêmes calculés avant les expressions non enchassées par des parenthèses.

   Si on ajoute d'autres niveaux de priorité (et nous en aurons besoin dans notre langage), une telle idée ne sera pas praticable.

Exercice 5: Expression et priorité des opérateurs

Dans cet exercice, nous allons voir comment utiliser le concept de précédence d'opérateurs (c'est-à-dire un ordre sur les opérateurs permettant de décrire l'associativité et la priorité des opérateurs) pour désambiguïser automatiquement les conflits dans une grammaire naturelle.

Cet exercice est à réaliser dans `td_4/eee_parser`.

Reprenons la grammaire produite au début de l'exercice précédent. Il y avait de nombreux conflits. Nous devrions les réintroduire petit à petit.

1. Dans `eee_parser/Parser.mly` ne gardez que les règles correspondant aux nombres et à l'addition (ignorez tous les autres opérateurs).

   Qu'observe-t-on dans le fichier `eee_parser.conflicts` ? Expliquer.

2. Modifier `eee_parser/Parser.mly` en prenant en compte que l'opérateur $+$ est associatif à gauche, c'est-à-dire que $a + b + c$ s'analyse sans ambiguïté $(a + b) + c$.

   Cela s'obtient en ajoutant la directive `%left ADD` dans l'entête du fichier (après la déclaration des tokens).

   Tester, il faut obtenir que `eee_parser.conflicts` soit vide.

   Observez que les arbres produits correspondent bien à une associativité à gauche.

3. Reprendre le fichier `eee_parser/Parser.mly` en ajoutant l'opérateur $\times$ et sans dire qu'il est associatif à gauche.

   Qu'observe-t-on dans le fichier `eee_parser.conflicts` ? Expliquer.

4. Modifier à nouveau `eee_parser/Parser.mly` en prenant en compte que tous les opérateurs sont associatifs à gauche, et que les opérateurs sur les facteurs sont prioritaires sur les opérateurs sur les termes.

   c'est-à-dire que $a + b \times c$ s'analyse sans ambiguïté $a + (b \times c)$.

   Pour cela, l'ordre des déclarations `%left` est important : les déclarations les plus hautes étant moins prioritaires (i.e., à l'extérieur).

   Tester, il faut obtenir que `eee_parser.conflicts` soit vide.

5. Reprendre le fichier `eee_parser/Parser.mly` en ajoutant les opérateurs unaires.

   Qu'observe-t-on dans le fichier `eee_parser.conflicts` ? Expliquer.

6. Modifier encore une fois `eee_parser/Parser.mly` en prenant en compte que l'opérateur unaire $-$ n'a pas les mêmes propriétés algébriques que l'opérateur binaire $-$. En effet, le second est associatif à gauche et non prioritaire sur la multiplication, alors que le premier n'a pas d'associativité (il opère sur l'élément juste à sa droite) et est prioritaire sur la multiplication:

   $a \times - - b = a \times (-(-b))$

   Vous aurez besoin de la directive `%nonassoc` pour cela. Vous aurez également besoin d'un token fictif (qui se déclare en lui donnant une priorité) et en associant sa priorité à la règle en ajoutant l'annotation `%prec USUB` (si le token fictif se nomme `USUB`) à la fin de la production (avant ou après l'action associée).

   Tester, il faut obtenir que `eee_parser.conflicts` soit à nouveau vide.

7. Utilisez la même technique pour désambiguïser la grammaire de `Dyck_simple` et ne plus avoir de conflit.

Sémantique

Exercice 6: Interpréteur d'expressions par un parseur

Nous avons maintenant un analyseur de la grammaire G_2 et il nous tarde d'y ajouter une sémantique. Nous nous doutons de la sémantique: il faudra interpréter les expressions arithmétiques pour en donner les valeurs numériques !

L'exemple suivant montre comment `Menhir` associe les différentes occurrences des symboles avec les attributs.

La règle de grammaire suivante:

```
e.result1 -> e.expression1 ADD e.expression2
e.result2 -> INT.expression

result1 = expression1 + expression2
result2 = expression
```

S'écrit ainsi en `Menhir` :

```
e:
| expression1 = e ADD expression2 = e          { expression1 + expression2 }
| expression = INT                             { expression }
```

Il existe une autre syntaxe supportée par Menhir, qui est utilisée notamment dans d'autres générateurs de parseurs (yacc, bison, ocamlyacc, etc), mais cette dernière est découragée par Menhir (pour des raisons de lisibilité). On la donne cependant ici, car elle est très diffusée et pour que vous l'ayez déjà vue.

```
e:
| e ADD e           { $1 + $3 }
| INT               { $1 }
```

1. Écrire la sémantique de la grammaire G_2 où l'ensemble des calculs sont réalisés sur des nombres entiers et le résultat est simplement affiché par le non-terminal main (avec `Format.printf`). Le non-terminal expression sera de type `int` et aura toutes ses productions associée à des valeurs `int`, le non-terminal main sera de type `unit` et affichera simplement la valeur de l'expression qu'il produit.

   Vous pourrez voir le résultat s'afficher en analysant une expression avec la grammaire Eee dans `parser_visualiser.out`. Une fois la visualisation quittée, vous verrez s'afficher le résultat du calcul.

Langage du cours

Exercice 7: Parseur du langage du cours Cet exercice est à effectuer dans le dossier `compiler/course_language`.

De plus, pour cet exercice, en plus de l'exécutable précédent, vous disposez aussi de l'exécutable `interactive_parser.out` qui vous permet à la fois de visualiser le parsing d'un fichier de code, mais également de l'exécuter ensuite (comme lors du TD2).

1. Remplissez le fichier `Parser.mly` avec la grammaire du langage du cours. Dans un premier temps, considérez que le `else` est obligatoire derrière un `then`.

   Le non-terminal `main` vous est fourni, ne le modifiez pas.

   Votre parseur ne doit générer aucun conflit.

   Votre parseur doit générer des programmes au sens de ceux que l'on a manipulé dans le TD2. La définition des types représentant les programmes se trouvent dans `language/ast.mli`.

   Vous allez devoir accepter des listes pour certaines constructions (appels de fonctions, déclarations de fonctions, et blocs). Pour ce faire, vous devez faire un non-terminal dédié qui créée la liste pas à pas. Dans ce cas, attention au sens où vous la construisez. Si elle n'est pas dans le bon sens, n'oubliez pas de la retourner avec List.rev dans les non-terminaux appelant. Vous pouvez également regarder la documentation de Menhir pour comprendre comment le faire directement dans une règle (mais attention, c'est une utilisation plus avancée qui peut vous générer des erreurs si vous ne le faites pas correctement).

   Vous pourrez contrôler votre résultat avec `parser_visualiser.out`, mais également avec `interpreter.out` qui permettra de lire un programme en entrée et de l'exécuter avec l'interpréteur du TD2 (que l'on vous fournit pour l'occasion).

   L'interpréteur est capable de supporter des expressions et des instructions seules (qu'il interprétera sur l'environnement vide). Vous pouvez donc concevoir votre grammaire au fur et à mesure en commençant par les expressions.

   Une fois cette grammaire implémentée, vous aurez un interpréteur complet du langage du cours.

   Un certain nombre de programmes exemples sont fournis dans le dossier `programs` (qui correspondent à une partie des exemples du TD2).

   Conseil : Il y a pas mal de redondances dans les opérations binaires. on peut limiter la redondance en utilisant les non-terminaux inlinés. À la compilation, ceux-ci sont simplement copiés et développés dans les règles qui les génèrent.

   Ainsi, si on écrit :

```
toto:
| e = toto s = tata {A(e,s)}

%inline tata:
| Add {"Add"}
| Sub {"Sub"}
```

   cela reviendra à la même chose que

```
toto:
| e = toto Add { A(e,"Add")}
| e = toto Sub { A(e,"Sub")}
```

   Cela peut vous permettre de rendre une grammaire plus lisible, sans avoir à la modifier (sans le `%inline`, on introduirait réellement un non-terminal, ce qui pour les opérations binaires, rendrait la gestion des priorités impossible).

2. Maintenant, ajoutez la possibilité d'avoir des `then` non suivis de `else`. Si vous le faites en ajoutant simplement une règle, vous allez obtenir un conflit (mais essayez !).

   Le problème est que si on écrit le code suivant :

```
if x = 3 then if y = 4 then x := 4; else y := 3;
```

   il est possible que le `else` soit sur le premier ou le second `if`, c'est donc un cas ambigu. Plusieurs techniques existent pour désambiguïser ce cas : certains langages explicitent la fin du if dans la syntaxe, d'autre modifient la grammaire pour forcer un choix.

   Nous allons prendre cette seconde option, en adoptant la convention (assez répandue) que le `else` se rattache au `if` le plus proche (donc ici au `if y = 4`).

   Pour cela, il y a deux choix (à nouveau) :

   Le premier consiste à modifier la grammaire en dupliquant le non-terminal `instruction`, de manière à ce que si on utilise la règle avec un `else`, alors la branche du `then` ne puisse pas contenir de `if` sans `else` (sauf à l'intérieur d'un bloc).

   Cela dit, si vous faites cette option directement, vous allez avoir beaucoup de duplication, alors que la plupart des instructions (toutes sauf les `if` et `while`) sont identiques dans les deux cas. Pour ne pas avoir de duplication, regroupez ces règles dans un autres non-terminal inliné, qui sera utilisé dans les deux version du non-terminal `instruction`.

   Le second choix consiste à mettre une nouvelle fois des priorités. Cela s'effectue en déclarant les terminaux `THEN` et `ELSE` comme non-associatifs et en les ordonnant correctement.

   Essayez les deux.