Compilation

TP 2 : Grammaire

C. Alias & G. Iooss

Partie I – Analyse descendante

Exercice 0. *Échauffement*

Calculez NULLABLE, FIRST et FOLLOW pour chaque lexèmes de la grammaire suivante:

$$\begin{array}{lcl} S & \rightarrow & uBDz \\ B & \rightarrow & Bv|w \\ D & \rightarrow & EF \\ E & \rightarrow & y|\epsilon \\ F & \rightarrow & x|\epsilon \end{array}$$

Exercice 1. *Implémentation LL(k)*

Récupérez le fichier `LL.tar.gz` et décompressez-le. Ce fichier comprend 2 classes (en plus du fichier `main.cpp` et du `Makefile`):

- `Token.h` décrit la classe des lexèmes (terminaux et non-terminaux) constituant une grammaire hors contexte. Cette classe comporte 2 constructeurs: `Token()` pour construire ϵ et `Token(string nLex, bool isTerm)` pour construire les autres lexèmes (en précisant s'ils sont terminaux ou non).
- `Grammar.h` décrit la classe des grammaires hors-contexte. Les arguments du constructeurs sont la liste des règles de la grammaire et l'axiome de départ de la grammaire. L'attribut `lexGram` sert à retenir tous les lexèmes qui interviennent à un moment ou un autre dans la grammaire. Les attributs `nullable`, `first` et `follow` ne valent rien à l'initialisation, mais sont calculés via la méthode `fst_follow`.
- Au passage, un `multimap` se manipule et se comporte exactement comme un `map` [1], à la différence que des redondances de clefs peuvent arriver.

Manip.

- Construisez la grammaire donnée pendant l'Exercice 0.
- Calculez les ensembles "Nullable", "First" et "Follow", puis affichez le tout pour vérifier la réponse de l'exercice 0[2]
- Calculez et affichez les directeurs des règles de la grammaire. Montrez clairement si oui (ou non) cette grammaire est LL(1).
- Modifiez (très simplement) la grammaire de l'exercice 0 afin que le même langage soit reconnu et que la grammaire devienne LL(1).
- Pour finir, calculez et affichez la table de transition de cette nouvelle grammaire.

[1] Référence pratique: http://www.cplusplus.com/reference/stl/map/

[2] Indice: l'opérateur « a été surchargé pour afficher ces maps sans à avoir taper 3 lignes de code à chaque fois qu'on veut observer le contenu

Partie II – Analyse ascendante

Exercice 2. *Échauffement*

On considère la grammaire sur $\Sigma = \{\text{if}, \text{then}, \text{else}, \text{inst}\}$:

$$\begin{array}{lcl} Z & \rightarrow & S\$ \\ S & \rightarrow & \text{if } E \text{ then } S \text{ else } S \\ S & \rightarrow & \text{if } E \text{ then } S \\ S & \rightarrow & \text{inst} \end{array}$$

Questions.

- Tracez l'automate LR(0)
- La grammaire est elle LR(0)? SLR(1)? LR(1)?

Exercice 3. *Bison*

Comme on le voit, il serait fastidieux de construire l'automate de la grammaire à chaque fois... Heureusement, il existe un outil, Bison, qui génére le code C d'un analyseur syntaxique à partir d'une grammaire.

A) Intéraction flex/bison

Récupérez le fichier `src0_if.tar.gz` et décompactez le dans votre répertoire. Ouvrez le fichier `parser.ypp`. Ce fichier contient la spécification de la grammaire d'entrée. Comme pour flex, ce fichier comporte trois parties séparées par le lexème $$:

- **La première partie** contient des déclarations:

  %{ ... %} (lignes 5 – 23) contient du code C à copier/coller au début de l'analyseur. Typiquement, la déclaration des variables utilisées dans les actions sémantiques (vues plus tard), ou le `#include` du fichier `.h` qui déclare la classe d'un attribut à construire. Il est possible d'y mettre du code C++ à condition d'utiliser l'extension `.ypp`. Autrement, on utilise l'extension `.y`.

  %union { ... } (lignes 30 – 33) énumère les types utilisables pour les attributs (vus plus tard). Ici, on ne s'autorise que `char*` pour les attributs, et on le nomme "`string`".

  %token (lignes 40 – 41) énumère les symboles terminaux utilisables dans la grammaire. En général, il s'agit de lexèmes (*token* en anglais) reconnus par un analyseur lexical généré par flex.

  %type (ligne 43) associe un attribut à un symbole (terminal ou non). Lorsqu'il s'agit d'un symbole terminal (comme ici), l'attribut est "remonté" par l'analyseur lexical. Il faut donc que l'analyseur syntaxique et l'analyseur lexical se mettent d'accord sur un format d'échange.

  %start (ligne 46) spécifie l'axiome de la grammaire.

- **La seconde partie (lignes 49 – fin)** déclare les règles de la grammaire. La déclaration des productions partant d'un non-terminal sont groupés et séparées par "`|`". Elles finissent par un "`;`".
- **La troisième partie (absente ici)** commence par %% et peut contenir, comme pour flex, du code à copier/collerà la fin de l'analyseur. Typiquement une fonction `main` pour débugger la grammaire.

Manip.

- **Complétez votre fichier** `.ypp` pour pouvoir analyser la grammaire donnée dans l'exercice précédent.
- **Compilez avec** `make`. On produit le code C de l'analyseur syntaxique avec la commande: `bison --defines=parser.h -o parser.cc parser.ypp`. Bison génère également le fichier `parser.h` avec toutes les déclarations de la première partie. **Ouvrez le fichier** `parser.h`. Ce fichier est utilisé par l'analyseur lexical pour nommer les lexèmes et les attributs (et ainsi assurer la communication entre les deux analyseurs).

- **Ouvrez le fichier** `lexer.l` et constatez le `#include "parser.h"` (ligne 7) ainsi que l'utilisation de TK_IF, TK_THEN, etc déclarés dans le fichier `.ypp`. Comment la chaîne d'un identificateur est elle transmise?

B) Résolution des conflits shift/reduce

Comme on l'a vu, cette grammaire produit des conflits irrésolubles par LALR(1). D'ailleurs bison le signale délicatement avec le message: `parser.ypp: conflicts: 1 shift/reduce`[3]. Certes. On aimerait avoir d'avantage d'infos pour corriger la grammaire (ou au moins comprendre ce qui ne va pas).

Manip.

- **Ouvrez le fichier Makefile** et ajoutez les options `--report=lookahead --report-file=bison_report` à la ligne de commande de bison.
- **Recompilez et ouvrez le fichier bison_report**. Il contient, entre autres, une représentation texte de l'automate LALR(1) de la grammaire avec des indications sur la résolution des conflits. **Allez à l'état 8**. Comment se comporte l'analyseur? En général les conflits shift/reduce sont inévitables sur les grosses grammaires. Il faudra vérifier que bison à fait les bons choix.

C) Résolution des conflits reduce/reduce

Là, il s'agit d'un vrai bug dans la grammaire qu'il faut impérativement corriger. En gros, un conflit reduce/reduce signifie qu'un mot admet deux interprétations syntaxiques différentes. Par exemple, la chaîne "`int*x`" pourrait être vue comme une déclaration de type ou comme une expression!

Manip.

- Récupérez le fichier **src1_stmt.tar.gz** et décompactez le dans votre répertoire. Etudiez la grammaire. D'où vient le conflit reduce/reduce? Comment bison le résout-il?

Exercice 4. *La grammaire du C*

Ou plutôt d'un sous-ensemble significatif. Récupérez le fichier **src2_parser.tar.gz** et décompactez le dans votre répertoire.

Manip.

- **Inspectez la grammaire** en vous aidant du fichier-exemple `tests/tree.c`. Repérez les catégories syntaxiques.
- **Compilez**. Combien de conflits shift/reduce? **Ouvrez le fichier bison_report** et analysez soigneusement chacun de ces conflits ainsi que la résolution proposée par bison.

Exercice 5. *Action!*

Jusqu'à présent, nos analyseurs sont des oracles passifs. Or, on aimerait exécuter du code à mesure que l'analyse syntaxique progresse pour analyser le programme et produire du code intermédiaire. C'est ce que permettent les *grammaires attribuées*. On associe à chaque production un morceau de code qui sera exécuté chaque fois que la production sera réduite. Ce morceau de code est appelé *action sémantique* et calcule des attributs des non-terminaux en jeu. Pour commencer, considérons la grammaire d'expressions ETF:

$$\begin{array}{lcl} Z & \rightarrow & E\$ \\ E & \rightarrow & E + T \\ E & \rightarrow & T \\ T & \rightarrow & T * F \\ T & \rightarrow & F \\ F & \rightarrow & id \\ F & \rightarrow & (E) \end{array}$$

[3] ou `parser.ypp: conflits: 1 décalage/réduction` si votre `yacc` est en français

Questions.

- Attribuez la grammaire pour évaluer l'expression arithmétique.
- Exécutez votre grammaire attribuée sur l'expression 1+(2*3). On appliquera une évaluation dynamique.
- Si on utilise un analyseur LR, dans quel ordre sont exécutées les actions sémantiques? A quel parcours d'arbre cela correspond-t-il?

Manip.

- **Récupérez le fichier src3_ETF.tar.gz** et décompactez le dans votre répertoire. **Ouvrez le fichier parser.ypp**. Dans bison, il n'y a qu'un seul attribut par symbole (terminal et non-terminal), on le déclare avec la construction %type (ligne 55). L'attribut du ième symbole du membre droit d'une production est récupéré par $i(ligne 68). L'attribut du non-terminal dérivé se note $$. Tous les attributs sont synthétisés. Ainsi une action consiste exclusivement à calculer $$ en fonction des $i. Ce $$ sera à son tour le $i d'une production, etc.
- **Complétez la grammaire** pour évaluer l'expression. **Faites afficher le numéro de chaque régle réduite**.
- On souhaite construire un arbre de syntaxe abstrait (AST, Abstract syntax Tree) à partir de l'expression en utilisant Ast.h/.cc. **modifiez votre grammaire en conséquence**.

Exercice 6. *Bonus: le retour du LL*

Questions.

- Calculez $First_2$, $Follow_2$ et $Directeur_2$ sur la grammaire de l'exercice 0, sachant que:
  - $First_k$ est l'ensemble des mots à k lettres qui peuvent commencer une dérivation.
  - $Follow_k$ est l'ensemble des mots à k lettres qui peuvent suivre un lexème
  - $Directeur_k$ est l'ensemble des mots à k lettres qui peuvent suivre l'application d'une règle.
- Trouvez un algorithme pour calculer ces ensembles (droit d'utiliser $First$ et $Follow$ dans le calcul).
- Généralisez à $First_k$, $Follow_k$ et $Directeur_k$.
- Si vous êtes (très) motivé, implémentez tout ça (sauf si vous trouvez que le TP-man code avec ses pieds, vous pouvez reprendre le code du LL).[4]

[4] Un Kinder Bueno est gracieusement offert à toute personne réussissant à avoir une implémentation correcte avant la fin des 2 heures de TP.