UE 4TIN603U – Compilation – Licence 3 – 2018-2019

TD3 - Analyse syntaxique grâce à Beaver

Les ressources pour cette feuille de td se trouvent à l'URL suivante:

`https://www.labri.fr/perso/clement/enseignements/compilation/public/td3`

1. **Analyser les expressions avec des automates ?**

   Un langage de programmation contient des *expressions* de toutes sortes, logiques, arithmétiques, de types, etc. Même l'écriture d'une structure de contrôle de type `if test then instruction else instruction` est une *expression*, car elle contient des opérateurs `if`, `else` et des instructions qui ne sont rien d'autre que des structures de contrôle.

   Il est aisé de se convaincre que toutes ces expressions reviennent à un mot bien parenthésé a^nXb^n où a^n correspond à des parenthèses ouvrantes et b^n à des parenthèses fermantes, et X un contenu également bien parenthésé, de sorte que chaque parenthèse fermante b correspond à une parenthèse ouvrante a et qu'il y a autant de parenthèses fermantes que de parenthèses ouvrantes.

   Le plus simple des langages que l'on peut imaginer qui correspond au degré zéro d'une *expression* est le langage $\{a^nb^n/n \geq 0\}$ sur l'alphabet $\{a, b\}$.

   (a) Montrer que si l'on écrit un automate A qui contient exactement n états, tout chemin de longueur supérieure à n contient nécessairement au moins deux fois le même état. Intuitivement, que cela veut-il dire concernant le langage $\{a^nb^n/n \geq 0\}$?

   (b) **Lemme de l'étoile (que nous admettons mais ne démontrons pas dans ce cours)**

   Si L est un langage rationnel, alors il existe un entier N tel que tout mot ω du langage plus grand que N se décompose en $\omega = xyz$ et tel que
   — $|xy| \leq N$, $|y| \neq 0$
   — Pour tout entier $i > 0$, $xy^iz \in L$

   Montrer en utilisant le lemme de l'étoile (cf plus haut) que si l'on suppose que $\{a^nb^n/n \geq 0\}$ est rationnel, alors on aboutit à une contradiction :
   — Soit le mot $\omega = a^nb^n$ plus grand que n (il fait deux fois n)
   — Il se décompose en $\omega = xyz$ tel que
     — $|xy| \leq n$, $|y| \neq 0$
       Que peut-on en conclure ?
     — Pour tout entier $i > 0$, $xy^iz \in \{a^nb^n/n \geq 0\}$
       Où se trouve la contradiction ?

2. **Grammaire $a^n b^n$ – Premier programme Beaver**

   Documentation de `Beaver` : `http://beaver.sourceforge.net/`

   — Depuis la page `https://sourceforge.net/projects/beaver/files/`, télécharger la version la plus récente de Beaver : `0.9.11` à l'heure où j'écris ces lignes.

   — Extraire les fichiers suivants dans le répertoire `lib` d'un nouveau projet java nommé `td3` :

   — `beaver-cc.jar`

   — `beaver-rt.jar`

   — `beaver-ant.jar`

   Ces bibliothèques contiennent les élements nécessaires au bon fonctionnement de `Beaver` : `cc` (*compiler compiler*) Pour le compilateur de compilateur, `rt` (*runtime*) pour les bibliothèques nécessaires lors du fonctionnement du programme produit et `ant` pour le bon fonctionnement de la commande `ant`.

   depuis la page `https://www.labri.fr/perso/clement/enseignements/compilation/public/td3` :

   Télécharger les fichiers suivants et les mettre dans leurs répertoires respectifs

   — `td3.2/build.xml`

   — `td3.2/parser/ParserAb.grammar`

   — `td3.2/src/Main.java`

   — `td3.2/src/ScannerAb.java`

   — `td3.2/data/input`

   Listing 1 – td3.2/parser/ParserAb.grammar

```
%class "ParserAb";

%terminals A, B, NEWLINE;

%goal S;

S = X NEWLINE;

X = A X B
        |
        ;
```

   Lancer le programme `Beaver` depuis `Eclipse` ou depuis un terminal avec la commande `ant parser`.

   Cette action va produire :

   — `td3.2/src/ParserAb.java`

   — `td3.2/src/Terminals.java`

   (a) Consulter l'ensemble des fichiers produits, puis compiler et exécuter avec la commande `ant make`.

   Cette action va produire l'ensemble des fichiers `*.class` dans `bin`, puis va exécuter le programme sur `data/input` pour produire `data/output`

   (b) Modifier `ScannerAb.java` et `ParserAb.grammar` pour que le programme lise un ensemble de lignes ω avec $\omega \in \{a^n b^n / n \leq 0\}$

(c) Modifier `ParserAb.grammar` pour que le programme affiche un petit message constant pour chacune des lignes.

3. **Interface avec JFlex**

Télécharger les fichiers suivants et les mettre dans leurs répertoires respectifs

— `td3.3/build.xml`
— `td3.3/scanner/ScannerAb.jflex`

(a) Compléter le répertoire `td3.3` pour compiler le programme en utilisant l'analyseur lexical créé par `Jflex`.

4. **Grammaire des expressions de la logique propositionnelle**

La langage L des expressions de la logique propositionnelle se définit ainsi :

— Constantes
  — $1 \in L$ (vrai)
  — $0 \in L$ (faux)
— Variables
  — $p \in L$ où p est le nom d'une variable propositionnelle
— Formule unaire Si E est une expression de la logique propositionnelle
  — $\neg E \in L$
— Formules binaires Si E et F sont des expressions de la logique propositionnelle
  — $(E \vee F) \in L$ (ou)
  — $(E \wedge F) \in L$ (et)

(a) Télécharger les fichiers suivants et les mettre dans leurs répertoires respectifs

— `td3.4a/build.xml`
— `td3.4a/data/input`

Écrire un analyseur lexical en `JFlex` qui reconnaît l'ensemble des constantes, les variables, les opérateurs et des parenthèses du langage des expressions de la logique propositionnelle.

Encodage Unicode hexadécimal	Caractère
`\u2228`	$\vee$
`\u2227`	$\wedge$
`\u00AC`	$\neg$

(b) On remarquera que l'écriture courante des expressions se passe parfois de parenthèses dans la mesure où nous savons que $\wedge$ est prioritaire sur $\vee$. Ainsi, $(p \vee (q \wedge r))$ pourra aussi s'écrire $p \vee q \wedge r$. Pour simplifier, nous pourrons dire qu'une expression est une disjonction de termes ou un terme unique, et que chaque terme est une conjonction de facteurs ou un facteur unique. Enfin un facteur est une variable propositionnelle, une constante ou encore une expression entière notée entre parenthèses ou la négation de cette expression.

Écrire un analyseur syntaxique en `Beaver` selon les spécifications données ci-dessus.

(c) Ajouter les numéros de lignes et de colonnes aux symboles de manière à ce qu'un affichage de ces informations soit réalisé en cas d'erreur de tokenisation ou de syntaxe.

(d) Ajouter du code à chaque règle de la grammaire pour afficher la règle réduite. Qu'observe-t-on ?