Bureau d'études
Automates et Théorie des Langages
Documents électroniques autorisés
1h30

1 Prélude

- Télécharger depuis moodle l'archive `source.tgz`
- Désarchiver son contenu avec la commande : `tar xzvf source.tgz`
- Vous obtenez un répertoire nommé `source`
- Renommer ce répertoire sous la forme `source-Nom1-Nom2` (en remplaçant `Nom1` et `Nom2` par vos noms dans l'ordre alphabétique. Par exemple, si vous êtes le binôme Jacques Requin et Pénélope Pieuvre, vous utiliserez la commande :
  `mv source source-Pieuvre-Requin`

2 Postlude

Lorsque la séance se termine à 17h30 (18h pour les étudiants bénéficiant d'un tiers-temps), vous devrez :

- Vérifier que les résultats de vos travaux sont bien compilables
- Créer une archive avec la commande : `tar czvf source_Xxx_Yyy.tgz source_Xxx_Yyy`
- Déposer cette archive sur moodle

3 Paquetages hiérarchiques contenant des interfaces

L'objectif du bureau d'étude est de construire deux analyseurs pour une version simplifiée d'un langage de description de paquetages hiérarchiques contenant des interfaces. Ceux-ci seront composés d'un analyseur lexical construit avec l'outil `ocamlex` et d'un analyseur syntaxique construit respectivement, en exploitant l'outil `menhir` pour générer l'analyseur syntaxique, et la technique d'analyse descendante récursive programmée en `ocaml` en utilisant la structure de monade.

Voici un exemple de paquetage hiérarchique contenant des interfaces :

```
package a {
   interface A { }
   package b {
      interface B extends a.A {
         void m( B, int, a.C);
      }
   }
   interface C {
      a.b.B f( boolean, a.A);
   }
}
```

Cette syntaxe respecte les contraintes suivantes :

- les terminaux sont les noms de paquetages, d'interfaces et de méthodes, les mots clés `package`, `interface`, `extends`, `int`, `boolean`, `void`, les accolades ouvrante `{` et fermante `}`, les parenthèses ouvrante `(` et fermante `)`, le point virgule `;`, la virgule `,` et le point `.` ;

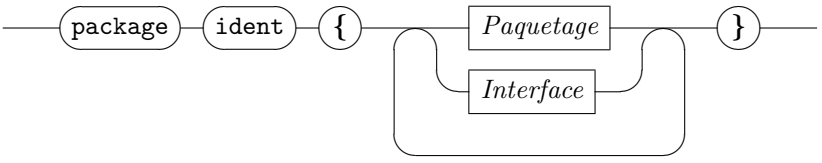
- la définition d'un paquetage débute par le mot clé `package` suivi d'un nom de paquetage, puis d'une suite d'éléments non vide comprise entre accolades ouvrante `{` et fermante `}` ;
- un élément est soit un paquetage, soit une interface ;
- une interface débute par le mot clé `interface` suivi du nom de l'interface éventuellement suivi d'extensions d'interfaces suivies d'une suite éventuellement vide de signatures de méthodes comprises entre accolades ouvrante `{` et fermante `}` ;
- les interfaces étendues débute par le mot clé `extends` suivi d'une liste non vide de noms qualifiés d'interfaces séparés par des virgules ;
- un nom qualifié d'interface est une suite éventuellement vide de noms de paquetages séparés par des points `.` suivie d'un nom d'interface ;
- une signature de méthode débute par le type de la valeur renvoyée par la méthode suivi du nom de la méthode, puis des types des paramètres de la méthode c'est-à-dire une suite éventuellement vide de types séparés par des virgules comprise entre parenthèses ouvrante `(` et fermante `)` ;
- un type est soit un type de base `boolean`, `int` ou `void`, soit un nom qualifié.

Voici les expressions régulières pour les terminaux complexes :

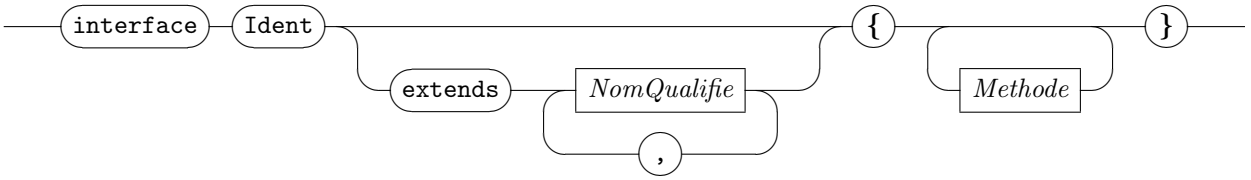
- nom d'interfaces (noté `Ident`) : $[A-Z][a-zA-Z]^\star$
- nom de paquetages et de méthodes (noté `ident`) : $[a-z][a-zA-Z]^\star$

Voici la grammaire au format graphique de Conway :

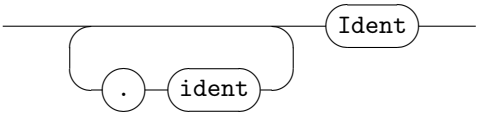
Paquetage



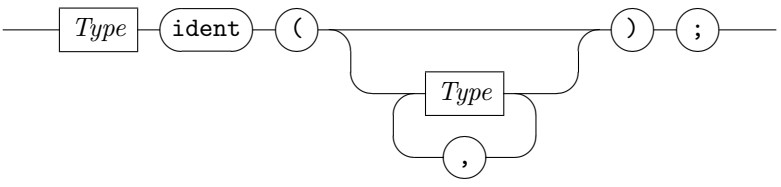
Interface



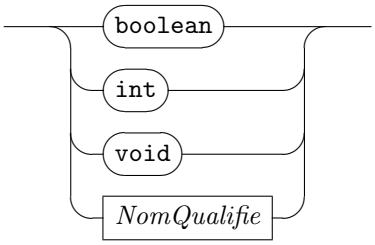
NomQualifie



Methode



Type



Voici une grammaire LL(1) sous la forme de règles de production et les symboles directeurs de chaque règle de production :

1.	$P \rightarrow$ `package` ***ident*** `{` $E\ S_E$ `}`	`package`
2.	$S_E \rightarrow \Lambda$	`}`
3.	$S_E \rightarrow E\ S_E$	`package` `interface`
4.	$E \rightarrow P$	`package`
5.	$E \rightarrow I$	`interface`
6.	$I \rightarrow$ `interface` ***Ident*** X `{` S_M `}`	`interface`
7.	$X \rightarrow \Lambda$	`{`
8.	$X \rightarrow$ `extends` Q ***Ident*** S_Q	`extends`
9.	$Q \rightarrow \Lambda$	***Ident***
10.	$Q \rightarrow$ ***ident*** `.` Q	***ident***
11.	$S_Q \rightarrow \Lambda$	`{`
12.	$S_Q \rightarrow$ `,` Q ***Ident*** S_Q	`,`
13.	$S_M \rightarrow \Lambda$	`}`
14.	$S_M \rightarrow M\ S_M$	`boolean` `int` `void` `ident` `Ident`
15.	$M \rightarrow T$ ***ident*** `(` O `)` `;`	`boolean` `int` `void` `ident` `Ident`
16.	$O \rightarrow \Lambda$	`)`
17.	$O \rightarrow T\ S_T$	`boolean` `int` `void` `ident` `Ident`
18.	$S_T \rightarrow \Lambda$	`)`
19.	$S_T \rightarrow$ `,` $T\ S_T$	`,`
20.	$T \rightarrow$ `boolean`	`boolean`
21.	$T \rightarrow$ `int`	`int`
22.	$T \rightarrow$ `void`	`void`
23.	$T \rightarrow Q$ ***Ident***	***ident Ident***

4 Analyseur syntaxique ascendant

Vous devez travailler dans le répertoire `ascendant`.

Vous compilerez régulièrement les modifications réalisées pour détecter les erreurs au plus tôt.

Vous testerez régulièrement votre travail en ajoutant des tests de difficulté croissante dans le répertoire `tests` à la racine de l'archive.

La sémantique de l'analyseur syntaxique consiste à afficher les règles appliquées pour l'analyse.

Complétez les fichiers `Lexer.mll` (analyseur lexical) puis `Parser.mly` (analyseur syntaxique). Le programme principal est contenu dans le fichier `MainPackage.ml`. La commande `dune build MainPackage.exe` produit l'exécutable `_build/default/MainPackage.exe` qui prend comme paramètre le fichier à analyser. L'exemple de ce sujet est disponible dans le répertoire `tests`.

5 Analyseur syntaxique par descente récursive

Vous devez travailler dans le répertoire `descendant`.

Vous compilerez régulièrement les modifications réalisées pour détecter les erreurs au plus tôt.

Vous testerez régulièrement votre travail en ajoutant des tests de difficulté croissante dans le répertoire `tests` à la racine de l'archive.

L'analyseur syntaxique devra afficher les règles appliquées au fur et à mesure de l'analyse. Les éléments nécessaires sont disponibles en commentaires dans le fichier.

Complétez les fichiers `Scanner.mll` (analyseur lexical) puis `Parser.ml` (analyseur syntaxique). Attention, le nom du fichier contenant l'analyseur lexical est différent de celui du premier exercice car les actions lexicales effectuées sont différentes (l'analyseur lexical du premier exercice renvoie l'unité lexicale reconnue; l'analyseur lexical du second exercice construit la liste de toutes les unités lexicales et renvoie cette liste). Le programme principal est contenu dans le fichier `MainPackage.ml`. La commande `dune build MainPackage.exe` produit l'exécutable `_build/default/MainPackage.exe` qui prend comme paramètre le fichier à analyser. L'exemple de ce sujet est disponible dans le répertoire `tests`.