Bureau d'études
Automates et Théorie des Langages
Documents 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 le nom des deux membres du binôme). Par exemple, si les membres sont Xavier Crégut et Marc Pantel, vous utiliserez la commande : `mv source source_Cregut_Pantel`

2 Postlude

Lorsque la séance se termine à 17h15 (17h45 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 Un langage de description de Machines à états (state machine)

L'objectif du bureau d'étude est de construire deux analyseurs pour une version simplifiée d'un langage de description de machines à états. 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 machine à états :

```
machine lamp {
   event switchOn
   event switchOff
   region bulb {
      state Off starts ends
      state On
   }
   from bulb.Off to buld.On on switchOn
   from buld.On to buld.Off on SwitchOff
}
```

Cette syntaxe respecte les contraintes suivantes :

- les terminaux sont les identificateurs `ident`, les mots clés `machine`, `event`, `region`, `state`, `from`, `to`, `on`, `starts`, `ends`, les accolades ouvrante `{` et fermante `}`, et le point `.` ;
- la définition d'une machine à états débute par le mot clé `machine` suivi d'un identificateur, le nom de la machine, puis d'une suite de composants éventuellement vide comprise entre accolades ouvrante `{` et fermante `}` ;

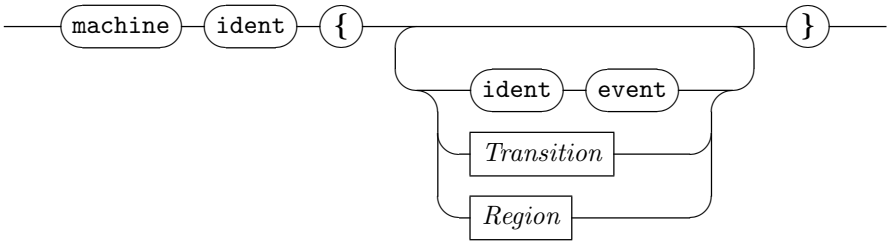
- un composant est soit un événement, soit une transition, soit une région ;
- un événement débute par le mot clé `event` suivi d'un identificateur, le nom de l'événement ;
- une transition débute par le mot clé `from` suivi du nom qualifié de l'état ou la région source puis du mot clé `to` suivi du nom qualifié de l'état ou la région cible puis du mot clé `on` suivi du nom de l'événement ;
- un nom qualifié est une suite non vide d'identificateurs séparés par des points `.` ;
- une région débute par le mot clé `region` suivi d'un identificateur, le nom de la région, puis d'une suite non vide d'états comprise entre accolades ouvrante `{` et fermante `}` ;
- un état débute par le mot clé `state` suivi d'un identificateur, le nom de l'état puis éventuellement du mot clé `starts` qui indique que l'état est initial, puis éventuellement du mot clé `ends` qui indique que l'état est terminal, éventuellement suivie d'une suite non vide de régions comprise entre accolades ouvrante `{` et fermante `}`.

Voici les expressions régulières pour les terminaux complexes :

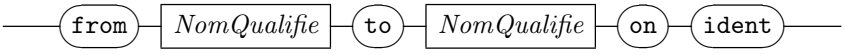
- `ident` : "$[A-Za-z][a-zA-Z]^\star$"

Voici la grammaire au format graphique de Conway :

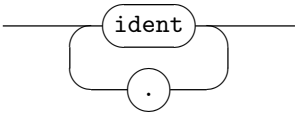
Machine



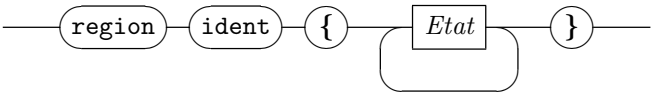
Transition



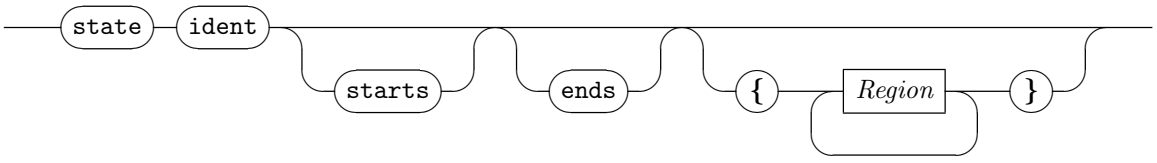
NomQualifie



Region



Etat



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

1.	$M \rightarrow$ `machine` ***ident*** `{` S_C `}`	`machine`
2.	$S_C \rightarrow \Lambda$	`}`
3.	$S_C \rightarrow C\ S_C$	`event` `from` `region`
4.	$C \rightarrow$ `event` ***ident***	`event`
5.	$C \rightarrow$ `from` N_Q `to` N_Q `on` ***ident***	`from`
6.	$N_Q \rightarrow$ ***ident*** S_Q	***ident***
7.	$S_Q \rightarrow \Lambda$	`to` `on`
8.	$S_Q \rightarrow$ `.` ***ident*** S_Q	`.`
9.	$C \rightarrow R$	`region`
10.	$R \rightarrow$ `region` ***ident*** `{` $E\ S_E$ `}`	`region`
11.	$S_E \rightarrow \Lambda$	`}`
12.	$S_E \rightarrow E\ S_E$	`state`
13.	$E \rightarrow$ `state` ***ident*** $E_S\ E_E\ E_C$	`state`
14.	$E_S \rightarrow \Lambda$	`ends` `{` `}` `state`
15.	$E_S \rightarrow$ `starts`	`starts`
16.	$E_E \rightarrow \Lambda$	`{` `}` `state`
17.	$E_E \rightarrow$ `ends`	`ends`
18.	$E_C \rightarrow \Lambda$	`}` `state`
19.	$E_C \rightarrow$ `{` $R\ S_R$ `}`	`{`
20.	$S_R \rightarrow \Lambda$	`}`
21.	$S_R \rightarrow R\ S_R$	`region`

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 `MainProlog.ml`. La commande `dune build MainMachine.exe` produit l'exécutable `_build/default/MainMachine.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 `MainMachine.ml`. La commande `dune build MainMachine.exe` produit l'exécutable `_build/default/MainMachine.exe` qui prend comme paramètre le fichier à analyser. L'exemple de ce sujet est disponible dans le répertoire `tests`.