Compilateur Java

première partie : analyse syntaxique

à rendre avant le lundi 3 novembre 10 heures

Le langage considéré dans ce projet est un sous-ensemble de Java, si bien que les fichiers pourront être compilés aussi bien par votre compilateur que par tout autre compilateur Java (et également exécutés, comme nous le verrons dans la troisième partie).

1 Analyse lexicale

Espaces, tabulations et retour-chariots constituent les blancs. Les commentaires peuvent prendre deux formes :

- à la C, débutant par `/*` et s'étendant jusqu'à `*/`, et ne pouvant être imbriqués ;
- à la C++, débutant par `//` et s'étendant jusqu'à la fin de la ligne.

Les identificateurs obéissent à l'expression régulière $\langle ident \rangle$ suivante :

$$\begin{array}{lcl} \langle \mathit{chiffre} \rangle & ::= & \texttt{0}\text{–}\texttt{9} \\ \langle \mathit{alpha} \rangle & ::= & \texttt{a}\text{–}\texttt{z} \mid \texttt{A}\text{–}\texttt{Z} \\ \langle \mathit{ident} \rangle & ::= & (\langle \mathit{alpha} \rangle \mid \texttt{\_})\ (\langle \mathit{alpha} \rangle \mid \langle \mathit{chiffre} \rangle \mid \texttt{\_})\star \end{array}$$

Les identificateurs suivants sont des mots clés :

```
boolean class double else extends false for if instanceof int
new null private return static true this void while
```

Les constantes entières obéissent à l'expression régulière $\langle entier \rangle$ suivante :

$$\langle \mathit{entier} \rangle \quad ::= \quad \texttt{0} \mid \texttt{1}\text{–}\texttt{9}\ \langle \mathit{chiffre} \rangle\star$$

et les constantes réelles à l'expression $\langle réel \rangle$ suivante :

$$\begin{array}{lcl} \langle \mathit{réel} \rangle & ::= & \langle \mathit{chiffre} \rangle+\ \texttt{.}\ \langle \mathit{chiffre} \rangle\star\ ((\texttt{e} \mid \texttt{E})\ (\texttt{-} \mid \texttt{+})?\ \langle \mathit{chiffre} \rangle+)? \\ & \mid & \texttt{.}\ \langle \mathit{chiffre} \rangle+\ ((\texttt{e} \mid \texttt{E})\ (\texttt{-} \mid \texttt{+})?\ \langle \mathit{chiffre} \rangle+)? \\ & \mid & \langle \mathit{chiffre} \rangle+\ (\texttt{e} \mid \texttt{E})\ (\texttt{-} \mid \texttt{+})?\ \langle \mathit{chiffre} \rangle+ \end{array}$$

Les chaînes de caractères sont délimitées par le caractère `"`. Elles peuvent contenir n'importe quels caractères, à l'exception de `"`, `\` et du retour-chariot. Ces trois caractères doivent être encodés (à l'intérieur d'une chaîne) par les séquences `\"`, `\\` et `\n`, respectivement.

⟨*fichier*⟩ ::= ⟨*decl_classe*⟩⋆ `EOF`
⟨*decl_classe*⟩ ::= `class` ⟨*ident*⟩ (`extends` ⟨*ident*⟩) ? `{` champ⋆ `}`
⟨*champ*⟩ ::= ⟨*variable*⟩ `;` | ⟨*constructeur*⟩ | ⟨*méthode*⟩

⟨*variable*⟩ ::= ⟨*modificateur*⟩⋆ ⟨*expr_type*⟩ ⟨*decl_variables*⟩
⟨*decl_variables*⟩ ::= ⟨*decl_variable*⟩ | ⟨*decl_variable*⟩ `,` ⟨*decl_variables*⟩
⟨*decl_variable*⟩ ::= ⟨*ident*⟩ | ⟨*ident*⟩ `=` ⟨*expr*⟩

⟨*constructeur*⟩ ::= ⟨*modificateur*⟩⋆ ⟨*ident*⟩ `(` ⟨*paramètres*⟩ ? `)` `{` ⟨*instruction*⟩⋆ `}`
⟨*méthode*⟩ ::= ⟨*modificateur*⟩⋆ (⟨*expr_type*⟩ | `void`) ⟨*ident*⟩
`(` ⟨*paramètres*⟩ ? `)` `{` ⟨*instruction*⟩⋆ `}`

⟨*paramètres*⟩ ::= ⟨*expr_type*⟩ ⟨*ident*⟩ | ⟨*expr_type*⟩ ⟨*ident*⟩ `,` ⟨*paramètres*⟩
⟨*modificateur*⟩ ::= `static` | `private`
⟨*expr_type*⟩ ::= `boolean` | `int` | `double` | ⟨*ident*⟩

FIG. 1 – Grammaire des fichiers Java

2 Analyse syntaxique

La grammaire des fichiers sources considérée est donnée figures 1 et 2. Le point d'entrée est le non-terminal ⟨*fichier*⟩.

Les associativités et précédences des divers opérateurs sont données par la table suivante, de la plus faible à la plus forte précédence :

opérateur	associativité
`=`	droite
`||`	gauche
`&&`	gauche
`==`, `!=`	gauche
`>`, `>=`, `<`, `<=`, `instanceof`	gauche
`+`, `-`	gauche
`*`, `/`, `%`	gauche
`+`, `-` (unaires), `++`, `--`, `!`, cast	droite
`.`	gauche

Telle qu'elle est donnée ici, cette grammaire n'est pas LALR(1). Il vous est donc demandé d'en comprendre les raisons et de lever les ambiguïtés de la même manière qu'un compilateur Java (le plus simple étant pour cela de procéder à quelques tests avec `javac`). On rappelle que vous devez fournir un fichier `ocamlyacc` ne contenant aucun conflit.

$$
\begin{array}{lcl}
\langle\textit{expr}\rangle & ::= & \langle\textit{entier}\rangle \mid \langle\textit{réel}\rangle \mid \langle\textit{chaîne}\rangle \mid \texttt{true} \mid \texttt{false} \\
& \mid & \texttt{this} \\
& \mid & \texttt{null} \\
& \mid & \texttt{(}\ \langle\textit{expr}\rangle\ \texttt{)} \\
& \mid & \langle\textit{accès}\rangle \\
& \mid & \langle\textit{accès}\rangle\ \texttt{=}\ \langle\textit{expr}\rangle \\
& \mid & \langle\textit{accès}\rangle\ \texttt{(}\ \langle\textit{l\_expr}\rangle\,?\ \texttt{)} \\
& \mid & \texttt{new}\ \langle\textit{ident}\rangle\ \texttt{(}\ \langle\textit{l\_expr}\rangle\,?\ \texttt{)} \\
& \mid & \texttt{++}\ \langle\textit{expr}\rangle \mid \texttt{--}\ \langle\textit{expr}\rangle \mid \langle\textit{expr}\rangle\ \texttt{++} \mid \langle\textit{expr}\rangle\ \texttt{--} \\
& \mid & \texttt{!}\ \langle\textit{expr}\rangle \mid \texttt{-}\ \langle\textit{expr}\rangle \mid \texttt{+}\ \langle\textit{expr}\rangle \\
& \mid & \langle\textit{expr}\rangle\ \langle\textit{opérateur}\rangle\ \langle\textit{expr}\rangle \\
& \mid & \texttt{(}\ \langle\textit{expr\_type}\rangle\ \texttt{)}\ \langle\textit{expr}\rangle \\
& \mid & \langle\textit{expr}\rangle\ \texttt{instanceof}\ \langle\textit{expr\_type}\rangle
\end{array}
$$

$$
\begin{array}{lcl}
\langle\textit{opérateur}\rangle & ::= & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{\&\&} \mid \texttt{||} \\
\langle\textit{accès}\rangle & ::= & \langle\textit{ident}\rangle \mid \langle\textit{expr}\rangle\ \texttt{.}\ \langle\textit{ident}\rangle \\
\langle\textit{l\_expr}\rangle & ::= & \langle\textit{expr}\rangle \mid \langle\textit{expr}\rangle\ \texttt{,}\ \langle\textit{l\_expr}\rangle
\end{array}
$$

$$
\begin{array}{lcl}
\langle\textit{instruction}\rangle & ::= & ; \\
& \mid & \langle\textit{expr}\rangle\ ; \\
& \mid & \langle\textit{expr\_type}\rangle\ \langle\textit{decl\_variables}\rangle\ ; \\
& \mid & \texttt{if}\ \texttt{(}\ \langle\textit{expr}\rangle\ \texttt{)}\ \langle\textit{instruction}\rangle \\
& \mid & \texttt{if}\ \texttt{(}\ \langle\textit{expr}\rangle\ \texttt{)}\ \langle\textit{instruction}\rangle\ \texttt{else}\ \langle\textit{instruction}\rangle \\
& \mid & \texttt{while}\ \texttt{(}\ \langle\textit{expr}\rangle\ \texttt{)}\ \langle\textit{instruction}\rangle \\
& \mid & \texttt{for}\ \texttt{(}\ \langle\textit{l\_expr}\rangle\,?\ ;\ \langle\textit{expr}\rangle\,?\ ;\ \langle\textit{l\_expr}\rangle\,?\ \texttt{)}\ \langle\textit{instruction}\rangle \\
& \mid & \texttt{for}\ \texttt{(}\ \langle\textit{expr\_type}\rangle\ \langle\textit{decl\_variables}\rangle\ ; \\
& & \qquad \langle\textit{expr}\rangle\,?\ ;\ \langle\textit{l\_expr}\rangle\,?\ \texttt{)}\ \langle\textit{instruction}\rangle \\
& \mid & \texttt{\{}\ \langle\textit{instruction}\rangle\star\ \texttt{\}} \\
& \mid & \texttt{return}\ \langle\textit{expr}\rangle\,?\ ;
\end{array}
$$

FIG. 2 – Grammaire des expressions et des instructions

3 Localisation des erreurs

Lorsqu'une erreur lexicale ou syntaxique est détectée par votre compilateur, elle doit être signalée le plus précisémment possible, par sa nature et sa localisation dans le fichier source. On adoptera le format suivant pour cette signalisation :

```
File "test.java", line 4, characters 5-6:
syntax error
```

L'anglicisme de la première ligne est nécessaire pour que la fonction `next-error` d'Emacs puisse interpréter la localisation et placer ainsi automatiquement le curseur sur l'emplacement de l'erreur. (En revanche, le message d'erreur proprement dit pourra être écrit en français si vous le souhaitez.)

Les localisations peuvent être obtenues grâce aux fonctions `Lexing.lexeme_start`, `Lexing.lexeme_end`, `Parsing.rhs_start`, `Parsing.rhs_end`. Ces fonctions retournent un entier, correspondant à une position absolue dans le fichier (c'est-à-dire un nombre de caractères depuis le début du fichier). Pour transformer cette position absolue en une position relative (ligne et colonne), on adoptera la méthode suivante : en cas d'erreur, on relit le fichier jusqu'à la position de l'erreur, en comptant le nombre de lignes et le nombre de caractères lus.

Anticipation. Dans la deuxième partie, les erreurs sémantiques doivent également être localisées avec précision, c'est-à-dire indiquer le sous-terme en cause. Pour cela il est donc nécessaire que l'arbre de syntaxe abstraite contienne cette information de localisation. Il vous est donc conseillé d'anticiper ce besoin et de construire dès à présent un arbre de syntaxe abstraite adéquat.

4 Travail demandé

Écrire un compilateur, appelons-le `jc`, acceptant sur sa ligne de commande exactement un fichier Java et éventuellement l'option `-parse-only`. Cette option est sans effet dans cette première partie et son utilisation sera justifiée par la suite.

Si le fichier est conforme à la syntaxe décrite dans ce document, le programme doit terminer avec le code de sortie 0 (`exit 0` explicite ou terminaison normale du programme), sans rien afficher. En cas d'erreur lexicale ou syntaxique, celle-ci doit être signalée (de la manière indiquée ci-dessus) et le programme doit terminer avec le code de sortie 1 (`exit 1`). En cas d'autre erreur (une erreur du compilateur lui-même), le programme doit terminer avec le code de sortie 2 (`exit 2`).

Modalités de remise de votre projet. Votre projet doit se présenter sous forme d'une archive tar compressée (option "z" de tar), appelée *vos_noms*`.tgz` qui doit contenir un répertoire appelé *vos_noms* (exemple : `dupont-durand.tgz`). Dans ce répertoire doivent se trouver les *sources* de votre programme (ne donnez pas les fichiers compilés). Quand on se place dans ce répertoire, la commande `make` doit créer votre compilateur, qui sera appelé `jc`. La commande `make clean` doit effacer tous les fichiers que `make` a engendré et ne laisser dans le répertoire que les fichiers sources.

Le projet est à faire impérativement en binôme. Chacune des trois parties qui le composent est à rendre à votre chargé de TP par e-mail[1] au plus tard à la date indiquée par le calendrier suivant :

Partie 1 Lundi 3/11 avant 10h00

Partie 2 Lundi 1/12 avant 10h00

Partie 3 Mardi 6/01 avant 10h00

[1] `sylvain.conchon@lri.fr`, `jean-christophe.filliatre@lri.fr`, `julien.forest@lri.fr`, `clement.renard@inria.fr`, `julien.signoles@lri.fr`, `julien.narboux@inria.fr`