Compilateur Pascal avec filtrage

première partie : analyse syntaxique

à rendre avant le lundi 17 octobre 10 heures

Le langage considéré dans ce projet est un sous-ensemble de Pascal (pas de pointeurs, pas de fonctions locales, etc.), augmenté de types algébriques et de filtrage à la Caml. Plus précisément, il s'agit des deux ajouts suivants :

– la possibilité de déclarer des types récursifs à la ML, tel que
```
type list = Nil | Cons of integer * list;
```
– la possibilité d'effectuer du filtrage sur les valeurs de ces types, grâce à une modification de l'instruction `case` :
```
case l of
  Nil:
     length := 0;
| Cons (_, r):
     length := length(r) + 1;
end;
```

Les fichiers n'utilisant pas cette extension pourront être compilés aussi bien par votre compilateur que par tout autre compilateur Pascal.

Dans la suite, nous utilisons les notations suivantes dans les grammaires :

$\langle$*règle*$\rangle^\star$	répétition de la règle $\langle$*règle*$\rangle$ un nombre quelconque de fois (y compris aucune)
$\langle$*règle*$\rangle^\star_t$	répétition de la règle $\langle$*règle*$\rangle$ un nombre quelconque de fois (y compris aucune), les occurrences étant séparées par le terminal t
$\langle$*règle*$\rangle^+$	répétition de la règle $\langle$*règle*$\rangle$ au moins une fois
$\langle$*règle*$\rangle^+_t$	répétition de la règle $\langle$*règle*$\rangle$ au moins une fois, les occurrences étant séparées par le terminal t
$\langle$*règle*$\rangle$?	utilisation optionnelle de la règle $\langle$*règle*$\rangle$ (i.e. 0 ou 1 fois)
($\langle$*règle*$\rangle$)	parenthésage ; attention à ne pas confondre ces parenthèses avec les terminaux `(` et `)`

1 Analyse lexicale

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

– débutant par `(*` et s'étendant jusqu'à `*)`, et ne pouvant être imbriqués ;
– débutant par `{` et s'étendant jusqu'à `}`, et ne pouvant être imbriqués.

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

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

Les identificateurs suivants sont des mots clés :

`AND`	`ARRAY`	`BEGIN`	`BOOLEAN`	`CASE`
`DO`	`DIV`	`ELSE`	`END`	`FALSE`
`FOR`	`FUNCTION`	`IF`	`INTEGER`	`MOD`
`NOT`	`OF`	`OR`	`PROCEDURE`	`PROGRAM`
`REAL`	`THEN`	`TO`	`TRUE`	`TYPE`
`VAR`	`WHILE`			

Pour les identificateurs et les mots-clés, la casse des caractères est indifférente : ainsi, `xy`, `Xy`, `xY` et `XY` représentent tous les même identificateur, et de même `begin`, `BEGIN` ou encore `BeGin` représentent tous le même mot-clé.

Les constantes entières obéissent à l'expression régulière ⟨*entier*⟩ suivante :

$$\langle\textit{entier}\rangle \quad ::= \quad \langle\textit{chiffre}\rangle^{+}$$

et les constantes réelles à l'expression ⟨*réel*⟩ suivante :

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

2 Analyse syntaxique

La grammaire des fichiers sources considérée est donnée figure 1. 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**
`<` `<=` `>` `>=` `=` `<>`	à gauche
`+` `-` `or`	à gauche
`*` `/` `div` `mod` `and`	à gauche
`+` (unaire) `-` (unaire)	—
`not`	—

3 Localisation des erreurs

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

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

$$
\begin{array}{lcl}
\langle\textit{fichier}\rangle & ::= & \texttt{program}\ \langle\textit{ident}\rangle\ \texttt{;}\ \langle\textit{types}\rangle\,?\ \langle\textit{vars}\rangle\,?\ \langle\textit{definition}\rangle^{\star}\ \langle\textit{bloc}\rangle\ \texttt{.}\ \texttt{EOF} \\
\\
\langle\textit{types}\rangle & ::= & \texttt{type}\ \langle\textit{typedef}\rangle+ \\
\langle\textit{typedef}\rangle & ::= & \langle\textit{ident}\rangle\ \texttt{=}\ \langle\textit{constructeur}\rangle^{+}_{\mid}\ \texttt{;} \\
\langle\textit{constructeur}\rangle & ::= & \langle\textit{ident}\rangle \mid \langle\textit{ident}\rangle\ \texttt{of}\ \langle\textit{type}\rangle^{+}_{*} \\
\langle\textit{vars}\rangle & ::= & \texttt{var}\ \langle\textit{decl\_var}\rangle^{+} \\
\langle\textit{decl\_var}\rangle & ::= & \langle\textit{ident}\rangle^{+}_{,}\ \texttt{:}\ \langle\textit{type}\rangle\ \texttt{;} \\
\langle\textit{definition}\rangle & ::= & \texttt{function}\ \langle\textit{ident}\rangle\ \langle\textit{paramètres}\rangle\,?\ \texttt{:}\ \langle\textit{type}\rangle\ \texttt{;}\ \langle\textit{vars}\rangle\,?\ \langle\textit{bloc}\rangle\ \texttt{;} \\
 & \mid & \texttt{procedure}\ \langle\textit{ident}\rangle\ \langle\textit{paramètres}\rangle\,?\ \texttt{;}\ \langle\textit{vars}\rangle\,?\ \langle\textit{bloc}\rangle\ \texttt{;} \\
\langle\textit{paramètres}\rangle & ::= & \texttt{(}\ \langle\textit{param}\rangle^{+}_{;}\ \texttt{)} \\
\langle\textit{param}\rangle & ::= & \texttt{var}\,?\ \langle\textit{ident}\rangle^{+}_{,}\ \texttt{:}\ \langle\textit{type}\rangle \\
\\
\langle\textit{type}\rangle & ::= & \texttt{boolean} \mid \texttt{integer} \mid \texttt{real} \mid \langle\textit{ident}\rangle \\
 & \mid & \texttt{array [}\ \langle\textit{entier}\rangle\ \texttt{..}\ \langle\textit{entier}\rangle\ \texttt{] of}\ \langle\textit{type}\rangle \\
\\
\langle\textit{constant}\rangle & ::= & \texttt{true} \mid \texttt{false} \mid \langle\textit{entier}\rangle \mid \langle\textit{réel}\rangle \\
\langle\textit{expr}\rangle & ::= & \langle\textit{constant}\rangle \\
 & \mid & \langle\textit{lvalue}\rangle \\
 & \mid & \langle\textit{ident}\rangle\ \texttt{(}\ \langle\textit{expr}\rangle^{+}_{,}\ \texttt{)} \\
 & \mid & \texttt{not}\ \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}\rangle\ \texttt{)} \\
\langle\textit{lvalue}\rangle & ::= & \langle\textit{ident}\rangle \\
 & \mid & \langle\textit{lvalue}\rangle\ \texttt{[}\ \langle\textit{expr}\rangle\ \texttt{]} \\
\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{div} \mid \texttt{mod} \mid \texttt{and} \mid \texttt{or} \\
\langle\textit{instruction}\rangle & ::= & \langle\textit{lvalue}\rangle\ \texttt{:=}\ \langle\textit{expr}\rangle \\
 & \mid & \texttt{if}\ \langle\textit{expr}\rangle\ \texttt{then}\ \langle\textit{instruction}\rangle \\
 & \mid & \texttt{if}\ \langle\textit{expr}\rangle\ \texttt{then}\ \langle\textit{instruction}\rangle\ \texttt{else}\ \langle\textit{instruction}\rangle \\
 & \mid & \texttt{while}\ \langle\textit{expr}\rangle\ \texttt{do}\ \langle\textit{instruction}\rangle \\
 & \mid & \texttt{for}\ \langle\textit{ident}\rangle\ \texttt{:=}\ \langle\textit{expr}\rangle\ \texttt{to}\ \langle\textit{expr}\rangle\ \texttt{do}\ \langle\textit{instruction}\rangle \\
 & \mid & \texttt{case}\ \langle\textit{expr}\rangle\ \texttt{of}\ \langle\textit{branche}\rangle^{+}_{\mid}\ \texttt{end} \\
 & \mid & \langle\textit{bloc}\rangle \\
 & \mid & \langle\textit{ident}\rangle\ \texttt{(}\ \langle\textit{expr}\rangle^{+}_{,}\ \texttt{)} \\
 & \mid & \langle\textit{ident}\rangle \\
\\
\langle\textit{bloc}\rangle & ::= & \texttt{begin}\ \langle\textit{instruction}\rangle^{\star}_{;}\ (\,;\,?)\ \texttt{end} \\
\\
\langle\textit{branche}\rangle & ::= & \langle\textit{pattern}\rangle\ \texttt{:}\ \langle\textit{instruction}\rangle^{\star}_{;}\ (\,;\,?) \\
\langle\textit{pattern}\rangle & ::= & \texttt{\_} \\
 & \mid & \langle\textit{constant}\rangle \\
 & \mid & \langle\textit{ident}\rangle \\
 & \mid & \langle\textit{ident}\rangle\ \texttt{(}\ \langle\textit{pattern}\rangle^{+}_{,}\ \texttt{)}
\end{array}
$$

Fig. 1 – Grammaire des fichiers Pascal

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_p`, `Lexing.lexeme_end_p`, `Parsing.rhs_start_pos`, et `Parsing.rhs_end_pos`.

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 `compilo`, acceptant sur sa ligne de commande exactement un fichier Pascal 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`).

Votre code devra comprendre (au moins) les fichiers suivants :

- `lexer.mll` : le lexeur ocamllex
- `ast.mli` : le(s) type(s) des arbres de syntaxe abstraite
- `parser.mly` : le parseur ocamlyacc
- `main.ml` : le fichier principal

Il est possible d'utiliser d'autres fichiers, si cela est justifié dans le rapport.

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 `compilo`. 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. Il doit être remis à votre chargé de TP par e-mail[1] au plus tard le lundi 17 octobre à 10h00.

[1] `sylvain.conchon@lri.fr`, `jean-christophe.filliatre@lri.fr`, `marc.pouzet@lri.fr`, `matthieu.sozeau@lri.fr`, `benjamin.monate@cea.fr`