Compilateur Pascal avec modules

première partie : analyse syntaxique

à rendre avant le lundi 12 octobre à 12h00

Le langage considéré dans ce projet est un sous-ensemble de Pascal (pas de pointeurs, pas de fonctions locales, etc.), augmenté d'un système de modules à la Caml.

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 :

$\langle chiffre \rangle$ ::= `0–9`
$\langle alpha \rangle$ ::= `a–z` | `A–Z`
$\langle ident \rangle$ ::= $\langle alpha \rangle$ ($\langle alpha \rangle$ | `_` | $\langle chiffre \rangle$)*

Les identificateurs suivants sont des mots clés :

```
AND       ARRAY     BEGIN     BOOLEAN     DIV         DO
ELSE      END       FALSE     FOR         IF          INTEGER
MOD       NOT       OF        OR          PROCEDURE   PROGRAM
THEN      TO        TRUE      VAR         WHILE       STRUCT
SIG       MODULE    OPEN
```

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 $\langle entier \rangle$ suivante :

$\langle entier \rangle$ ::= $\langle chiffre \rangle^+$

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
```

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`).

$\langle fichier \rangle$ ::= `program` $\langle ident \rangle$ `;` $\langle définition \rangle^\star$ $\langle bloc \rangle$ `.` `EOF`

$\langle définition \rangle$ ::= $\langle vars \rangle$
| $\langle decl\_proc \rangle$ $\langle vars \rangle ?$ $\langle bloc \rangle$ `;`
| `module` $\langle ident \rangle$ $\langle params\_mod \rangle ?$ `(` `:` $\langle signature \rangle)?$ `:=` $\langle expr\_mod \rangle$ `;`
| `open` $\langle lident \rangle$ `;`

$\langle vars \rangle$ ::= `var` $\langle decl\_var \rangle^+$
$\langle decl\_var \rangle$ ::= $\langle ident \rangle^+_{,}$ `:` $\langle type \rangle$ `;`
$\langle decl\_proc \rangle$ ::= `procedure` $\langle ident \rangle$ $\langle params \rangle ?$ `;`
$\langle signature \rangle$ ::= `sig` $\langle déclaration \rangle^\star$ `end`
$\langle déclaration \rangle$ ::= $\langle vars \rangle$ | $\langle decl\_proc \rangle$
| `module` $\langle ident \rangle$ $\langle params\_mod \rangle ?$ `:` $\langle signature \rangle$ `;`
$\langle params \rangle$ ::= `(` $\langle param \rangle^+_{;}$ `)`
$\langle param \rangle$ ::= `var` $?$ $\langle ident \rangle^+_{,}$ `:` $\langle type \rangle$
$\langle params\_mod \rangle$ ::= `(` $\langle param\_mod \rangle^+_{;}$ `)`
$\langle param\_mod \rangle$ ::= $\langle ident \rangle$ `:` $\langle signature \rangle$

$\langle expr\_mod \rangle$ ::= `struct` $\langle définition \rangle^\star$ $(\langle bloc \rangle$ `;` $)?$ `end`
| $\langle lident \rangle$ `(` $\langle lident \rangle^+_{,}$ `)`

$\langle type \rangle$ ::= `boolean` | `integer`
| `array` `[` $\langle entier \rangle$ `..` $\langle entier \rangle$ `]` `of` $\langle type \rangle$

$\langle constant \rangle$ ::= `true` | `false` | $\langle entier \rangle$
$\langle expr \rangle$ ::= $\langle constant \rangle$ | $\langle lvalue \rangle$
| `not` $\langle expr \rangle$ | `-` $\langle expr \rangle$ | `+` $\langle expr \rangle$
| $\langle expr \rangle$ $\langle opérateur \rangle$ $\langle expr \rangle$
| `(` $\langle expr \rangle$ `)`
$\langle lident \rangle$ ::= $\langle ident \rangle$ | $\langle lident \rangle$ `.` $\langle ident \rangle$
$\langle lvalue \rangle$ ::= $\langle lident \rangle$ | $\langle lvalue \rangle$ `[` $\langle expr \rangle$ `]`
$\langle opérateur \rangle$ ::= `=` | `<>` | `<` | `<=` | `>` | `>=`
| `+` | `-` | `*` | `/` | `div` | `mod` | `and` | `or`
$\langle instruction \rangle$ ::= $\langle lvalue \rangle$ `:=` $\langle expr \rangle$
| `if` $\langle expr \rangle$ `then` $\langle instruction \rangle$
| `if` $\langle expr \rangle$ `then` $\langle instruction \rangle$ `else` $\langle instruction \rangle$
| `while` $\langle expr \rangle$ `do` $\langle instruction \rangle$
| `for` $\langle lident \rangle$ `:=` $\langle expr \rangle$ `to` $\langle expr \rangle$ `do` $\langle instruction \rangle$
| $\langle bloc \rangle$
| $\langle lident \rangle$ `(` $\langle expr \rangle^+_{,}$ `)`
| $\langle lident \rangle$

$\langle bloc \rangle$ ::= `begin` $\langle instruction \rangle^\star_{;}$ $(\ ;\ ?)$ `end`

FIG. 1 – Grammaire des fichiers Pascal

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 répertoire doit également contenir un rapport d'une page ou deux au format PDF.

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 12 octobre à 12h00.

[1] `francois.bobot@lri.fr`, `dephine.longuet@lri.fr`, `louis.mandel@lri.fr`