15 mars 2017

TP de Langages Formels

> Le TP est à rendre le mercredi 22 mars. Vous devez m'envoyer à l'adresse `marie.fortin@lsv.fr` une archive contenant votre code ainsi qu'un court compte-rendu.

Dans ce TP nous allons pratiquer deux outils importants issus du domaine des langages formels, couramment utilisés en partie frontale des compilateurs et interprètes de nos langages de programmation. L'analyse syntaxique est traditionnellement séparée en deux parties :

1. **Analyse lexicale** : extraire, à partir d'un fichier ou autre succession de caractères, une succession de symboles terminaux, plutôt appelés lexèmes ou tokens dans ce contexte. Par exemple, étant donné la suite de 5 caractères `"toto␣+␣1"` on cherchera à obtenir la succession de trois lexèmes `VAR("toto")`, `PLUS` et `INT(1)`.
   On notera déja une subtilité : les lexèmes prenent des arguments dans un domaine fini et semblent donc être en nombre infini. Nous verrons cependant que ces arguments ne pourront jouer un rôle dans les diverses analyses, ce qui justifie qu'on puisse parler de langages rationnels ou algébriques sur un alphabet fini comme dans le cours.
2. **Analyse syntaxique** : Analyser la succession de lexèmes en fonction d'une grammaire algébrique, pour en extraire un arbre syntaxique ou toute autre construction dérivée de cet arbre. Par exemple, dans un langage d'expressions arithmétiques raisonnable, notre suite de lexèmes sera acceptée et transformée en `Plus(Var("toto",Int(1)))`.

Pour l'analyse lexicale il existe une famille d'outils très proches, basés sur les expressions régulières, nommés `lex` de façon générique. Nous utiliserons `ocamllex` dans ce TP. Pour l'analyse syntaxique on parle de la famille des `yacc` : *yet another compiler compiler*, puisque l'outil génère à partir d'une grammaire algébrique un compilateur, ou du moins une partie d'un compilateur. Nous utiliserons `ocamlyacc`.

Pour commencer le TP, récupérez la base de code à l'adresse suivante :

`http://lsv.fr/~fortin/langages/tp.tar.gz`.

En décompressant l'archive vous devriez obtenir un répertoire `tp` dans lequel vous pourrez lancer `make` pour compiler le projet. Vous modifierez *uniquement* le fichier `lex.mll`, puis `parse.mly`, même s'il est bien sûr encouragé d'aller regarder l'ensemble des fichiers. Vous pouvez ensuite exécuter, par exemple `minirien '42+x'`, ce qui provoque l'affichage de l'expression construite à l'issue des deux analyses. *Attention* : le code donné initialement est *très* restreint, il n'acceptera presque aucune autre entrée que cet exemple !

1 Notre langage : mini-rien®

Nous travaillerons sur un langage totalement ad-hoc et inutile. La syntaxe de ses expressions e est donnée par une grammaire algébrique, comme on peut le trouver pour de nombreux langages de programmation :

$$\begin{array}{rcl} \textit{expr} & ::= & \textit{entier} \mid \textit{chaîne} \mid \textit{variable} \mid (\textit{expr}) \\ & \mid & \mathsf{let}\ \textit{variable} = \textit{expr}\ \mathsf{in}\ \textit{expr} \mid \textit{expr}\ \textit{infixop}\ \textit{expr} \mid \mathsf{not}\ \textit{expr} \end{array}$$

Ci-dessus, les éléments en *italique* sont des non-terminaux, non spécifiés pour l'instant sauf pour *expr*. La distinction entre analyse lexicale et syntaxique n'est volontairement pas faite à ce stade : nous ne précisons pas lesquels de ces non-terminaux seront des terminaux pour l'analyse syntaxique.

Les symboles '(', ')' et '=' sont donnés comme des terminaux, tout comme les mots-clés du langage let, in, et not. Ce sera effectivement le point de vue de l'analyseur syntaxique. On peut aussi bien sûr les voir comme des non-terminaux triviaux correspondant aux langages singleton correspondants : c'est en quelque sorte le point de vue du lexer.

Les **noms de variables** autorisés sont donnés comme dans le manuel OCaml [1], par une expression régulière notée de façon non-standard :

$$\textit{variable} ::= (a \ldots z \mid \_)\{a \ldots z \mid A \ldots Z \mid 0 \ldots 9 \mid \_ \mid\ '\}$$

La notation $\{\ldots\}$ dénote la possibilité d'inclure une nombre arbitraire de caractères décrits dans le corps de l'expression. Le reste des notations devrait être intuitif. Par exemple, `x2`, `_A1` et `toto'` sont des noms de variables autorisés.

Nous précisons maintenant la syntaxe des **entiers**, encore directement inspirée de ce qu'on trouve pour OCaml [2] :

$$\begin{array}{rcl} \textit{entier} & ::= & (0 \ldots 9)\{0 \ldots 9 \mid \_\} \\ & \mid & (0x \mid 0X)(0 \ldots 9 \mid A \ldots F \mid a \ldots f)\{0 \ldots 9 \mid A \ldots F \mid a \ldots f \mid \_\} \end{array}$$

Ici la notation $[c]$ dénote la possibilité, mais pas l'obligation, d'inclure le caractère c. Par exemple, l'entier 42 peut être écrit `42`, `4_2`, `0x0_2a`, etc.

Pour les **chaînes de caractères**, nous exigeons qu'elles soient commencées et terminées par le caractère `"`. Elles peuvent contenir tout caractère. Pour inclure un caractère `"` dans un chaîne de caractère, on donne la possibilité d'écrire `\"`. Par exemple, `"il␣dit␣\"oui\""` devra être accepté comme une chaîne de 12 caractères.

Enfin, les opérateurs infixes autorisés sont donnés ci-dessous :

$$\begin{array}{rcl} \textit{infixop} & ::= & \mathsf{and} \mid \mathsf{or} \mid \textit{fst\_opsymb}\ \textit{opsymbs} \\ \textit{fst\_opsymb} & ::= & (= \mid < \mid > \mid \hat{} \mid + \mid - \mid * \mid / \mid \%) \\ \textit{opsymbs} & ::= & \{= \mid < \mid > \mid \hat{} \mid + \mid - \mid * \mid / \mid \% \mid ! \mid \$ \mid ? \mid . \mid : \mid ;\} \end{array}$$

1. http://caml.inria.fr/pub/docs/manual-ocaml/lex.html#lowercase-ident
2. http://caml.inria.fr/pub/docs/manual-ocaml/lex.html#integer-literal

On pourra donc écrire `(1+1)`, `(1 /. 12)`, `(y <=> x)`, ou encore `(12 %%! x) <<>(2 --> 3)`. Les précédences entre opérateurs seront déterminées en fonction du premier symbole de l'opérateur. Nous aborderons ce point en deuxième partie. Pour l'instant, indiquons seulement que les quatre premiers symboles définissent des opérateurs de *comparaison*, les deux suivant des opérateurs *additifs* et les trois derniers des opérateurs *multiplicatifs*.

2 Analyse lexicale

Le fichier `lex.mll` définit un (ou plusieurs) analyseurs lexicaux. Dans le fichier fourni, cet analyseur est nommé `token` et défini par la syntaxe `rule token = ...`. La définition de l'analyseur lexical est donnée par une succession de règles, chacune étant composée d'une expression régulière et d'une *production* décrivant comment réagir quand l'expression est reconnue. La sémantique de l'analyseur est donnée par deux principes :

1. Une règle n'est activée que sur des reconnaissances maximales. Étant donné une succession de caractères en entrée, la règle est activée sur la reconnaissance d'un préfixe maximal du mot par une des expressions régulières de l'analyseur.

   Par exemple, si on a une règle qui n'accepte que des mots de longueur paire, le lexème *abcd* sera produit sur l'entrée *abcde*.

2. Quand deux règles peuvent être activées d'après le principe précédent, la priorité est donnée à la plus haute, c'est à dire celle écrite le plus haut dans la définition de l'analyseur.

   Par exemple, si l'on a une règle qui accepte *abcd* et qui a la priorité sur la règle précédente reconnaissant tous les mots de longueur paire, la règle prioritaire s'appliquera. Cela n'a un intérêt que si les productions des deux règles sont distinctes.

La syntaxe des expressions régulières est propre à `ocamllex`. Notez qu'on peut définir des variables correspondant à des expressions intermédiaires, ce qui facilite l'écriture et la lecture du code. Je vous invite à vous référer au manuel pour la syntaxe des expressions régulières : `http://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html`.

Les productions des règles définissent, la plupart du temps, quel lexème doit être retourné quand la règle est activée. Les lexèmes sont définis dans le fichier `parse.mly`, avec la grammaire du langage. On les retrouve aussi dans le fichier généré `parse.mli`. Une production peut aussi faire un appel récursif à l'analyseur : c'est typiquement utilisé pour les caractères d'espacement qu'on veut ignorer.

Question 1. Étant donné les deux règles qui régissent la sémantique de notre analyseur, qu'est ce qui ne va pas dans le fichier de départ ? Sur quelles entrées le voit-on ?

Question 2. Dans le fichier `lex.mll`, définissez l'expression régulière complète pour les variables, correspondant au lexème `VAR`. Tester en exécutant dans le shell, par exemple, `minirien 'toto'` ou `minirien "x'+x"`.

Question 3. De même pour les entiers (lexème `INT`). Bonne nouvelle : la fonctionalité standard `int_of_string` interprète la syntaxe utilisée, par exemple `int_of_string "0x2_a"= 42`.

Question 4. Idem pour les chaînes de caractères (lexème `STRING`).

Question 5. Et enfin pour les opérateurs, en produisant un lexème `BIN_CMP`, `BIN_PLUS` ou `BIN_MULT` en fonction du premier symbole de l'opérateur.

Question 6. Bonus. Le prototype renvoie des messages d'erreur assez mauvais pour les entrées multi-lignes. Améliorez cela en modifiant la règle qui gère les espacements pour mettre à jour les informations d'analyse lexicale quand un retour à la ligne est rencontré. Vous aurez à modifier `lexbuf.lex_curr_p` et notamment ses sous-champs `pos_bol` (indiquant la position du début de ligne) et `pos_lnum` (numéro de ligne), par exemple ainsi :

```
lexbuf.lex_curr_p <- {
  lexbuf.lex_curr_p with
    pos_bol = lexbuf.lex_curr_p.pos_cnum ;
    pos_lnum = 1 + lexbuf.lex_curr_p.pos_lnum }
```

Question 7. Bonus. Ajoutez la possibilité d'insérer des commentaires : comme en OCaml, on les délimitera `(* ainsi *)` et on supportera les commentaires imbriqués bien parenthésés, en acceptant par exemple `(** (* ( *) * *)` mais pas `(* (* *)`. Ce n'est bien sûr pas faisable avec une seule expression régulière, mais vos analyseurs lexicaux sont des fonctions récursives, avec état !

3 Analyse syntaxique

Pour la deuxième partie, nous allons travailler sur le fichier `parse.mly` qui définit l'analyseur syntaxique. Dans ce fichier, on trouve principalement la définition d'une grammaire dont chaque règle est enrichie d'une production qui indique ce que l'analyseur doit construire quand il fait une réduction de cette règle. Ici, on construit un arbre syntaxique épuré.

Question 8. Pourquoi votre analyseur ne fonctionne-t-il pas sur l'entrée `1=2` alors qu'il fonctionne sur `1<2` ? Réglez ce problème en ajoutant une règle dans la grammaire.

Testez ensuite l'analyseur sur des entrées comme `1+1+1`, `1+1*1`, `1*1+1`. Son comportement est-il conforme à la grammaire ? est-il satisfaisant ?

Les exemples précédents sont des situations de conflit. Pour les comprendre, ouvrez le fichier `parse.output`, produit lors de la compilation à chaque appel de `ocamlyacc -v parse.mly`. Vous devriez reconnaître les concepts vus en cours. Identifiez le(s) conflit(s) associé(s) aux exemples précédents (cherchez les occurrences du mot `conflict`).

Pour guider la résolution des conflits lors de la construction de l'analyseur, vous disposez des déclarations suivantes :

```
%left     TOKEN1
%right    TOKEN2
%nonassoc TOKEN3
```

Celles-ci indiquent, en gros, l'associativité d'un token. L'ordre relatif de ces indications détermine aussi la précédence entre tokens : ici, `TOKEN3` lie plus fortement que `TOKEN2`, qui lie plus fortement que `TOKEN1`. Ces notions correspondent en fait plus précisemment à des choix faits pour la résolution de conflits shift/reduce entre les différents tokens. Pour la description précise, voir le manuel :
`http://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html#sec288`

Question 9. Insérez, après la déclaration des tokens, les bonnes déclarations d'associativité et précédence pour les tokens `BIN_PLUS` et `BIN_MULT`, afin d'obtenir un comportement satisfaisant sur les exemples précédents.

Question 10. Faites de même pour `BIN_CMP`, `EQUALS`, `AND`, `OR` et `NOT`. On veut que

```
not 1+1 < 2 and 1 = 1 or foo
```

donne

```
or(and(not(<(+(1,1),2)),=(1,1)),foo).
```

Question 11. Il devrait vous rester un conflit shift/reduce. Analysez `parse.output` pour comprendre d'où il provient. Donnez un exemple d'entrée ambigue. Réglez le problème via une indication `%nonassoc` pour le token le plus à droite dans la règle réductible dans le conflit.

Nous nous proposons maintenant d'étendre mini-rien® avec un opérateur `case`, selon la syntaxe suivante :

$$\begin{array}{rcl} \mathit{expr} & ::= & \cdots \mid \mathsf{case}\ \mathit{expr}\ \mathsf{of}\ \mathit{pmatch} \\ \mathit{pmatch} & ::= & \mathit{prule} \mid \mathit{pmatch}\ \mathsf{|}\ \mathit{prule} \\ \mathit{prule} & ::= & \mathit{pat}\ \mathsf{=>}\ \mathit{expr} \end{array}$$

Les symboles '`case`', '`of`', '`=>`', et '`|`' (à ne pas confondre avec le séparateur '|' : on a deux règles $\mathit{pmatch} \to \mathit{prule}$, et $\mathit{pmatch} \to \mathit{pmatch}\ \mathsf{|}\ \mathit{prule}$) sont des terminaux.

Le non-terminal *pat* pourra correspondre à une constante (chaîne de caractère ou entier), une variable, ou '_'.

Question 12. Réalisez cette extension :

1. Ajoutez des tokens pour '`case`', '`of`', '`=>`', et '`|`'.

2. Ajoutez des règles pour produire ces tokens dans l'analyseur lexical.
3. Ajoutez des règles grammaticales pour accepter les expressions de la forme décrite au dessus. L'analyse de `case x + y of 0 => 0 | _ => 1` donnera par exemple `case(+(x,y),[(Int 0,Int 0),(Any,Int 1)])`. Éliminez les éventuels conflits introduits.

Question 13. On souhaite maintenant permettre le signe `-` unaire, pour écrire des choses comme `-x+2` ou `x * - y`.

1. Modifiez les deux analyseurs pour faire ceci : il ne va plus être possible de traiter le signe `-` comme un `BIN_PLUS` quelconque.
2. Vous ne devriez pas avoir de conflits, car votre règle nouvelle règle aura la précédence du non-terminal associé au signe `-` : le conflit est résolu silencieusement d'après les priorités que nous avons déja assignées entre les différents opérateurs binaires. Montrer que cela ne correspond pas à ce que l'on veut.
3. Corriger cela en créant un nouveau token pour désigner la priorité de la règle (cf. notation `%prec TOK`) et en donnant la bonne précédence à ce token.

Question 14. Quelques bonus possibles :

— Faites en sorte que l'analyseur syntaxique produise un AST dans lequel les calculs constants sont déja évalués. Par exemple, `1+(2*3)+x` devra donner `+(7,x)`.

— Détectez, dans l'analyseur syntaxique, l'utilisation de variables non déclarées. En d'autres termes, on veut rejeter les expressions comportant des variables libres.

— Une extension politiquement incorrecte pour la route ? On propose d'ajouter, comme dans certains Lisp, le crochet fermant qui ferme toutes les parenthèses ouvertes. On acceptera ainsi `(1]`, `((1]`, mais pas `1]`, `((1])`, ou `((1]+2)`.