22 mars 2018

TP de Langages Formels

> À rendre au plus tard le jeudi 29 mars à 23:59. Vous devez m'envoyer à l'adresse `marie.fortin@lsv.fr` une archive contenant vos fichiers ainsi qu'un court compte-rendu précisant les réponses aux questions et l'évolution du code quand nécessaire.

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 infini 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/langages2017/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

`parse 'y := 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 entrée !

1 Notre langage

Nous travaillerons sur un langage très simple, dont les commandes et les expressions sont données par la grammaire suivante :

$$\begin{array}{rcl} \mathit{cmd} & ::= & \mathsf{skip} \mid \mathit{variable} := \mathit{expr} \mid \mathit{cmd}\,;\,\mathit{cmd} \mid \mathsf{if}\ \mathit{expr}\ \mathsf{then}\ \mathit{cmd}\ \mathsf{else}\ \mathit{cmd} \\ & \mid & \mathsf{while}\ \mathit{expr}\ \mathsf{do}\ \mathit{cmd} \mid \mathsf{begin}\ \mathit{cmd}\ \mathsf{end} \\ \mathit{expr} & ::= & \mathit{entier} \mid \mathit{chaîne} \mid \mathit{variable} \mid (\mathit{expr}) \\ & \mid & \mathit{expr}\ \mathit{infixop}\ \mathit{expr} \mid \mathsf{not}\ \mathit{expr} \end{array}$$

Ci-dessus, les éléments en *italique* sont des non-terminaux, non spécifiés pour l'instant sauf pour *cmd* et *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 skip, if, then, else, while, do, begin, end, 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 correspondant : c'est en quelque sorte le point de vue du lexer.

Les **noms de variables** autorisés sont tous les mots formés à partir des lettres $a \ldots z$, $A \ldots Z$, des chiffres $0 \ldots 9$, et du caractère '_', et commençant soit par une lettre (minuscule ou majuscule), soit par '_'. Par exemple, `x2`, `_A1` et `Toto` sont des noms de variables autorisés.

Nous précisons maintenant la syntaxe des **entiers**, décrits par l'expressions régulière suivante :

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

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. De même, on pourra écrire `\\` mais pas `\`.

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

$$\begin{array}{rcl} \mathit{infixop} & ::= & \mathsf{and} \mid \mathsf{or} \mid \mathit{plus\_op} \mid \mathit{mult\_op} \mid \mathit{cmp\_op} \\ \mathit{plus\_op} & ::= & + \mid - \\ \mathit{mult\_op} & ::= & * \mid / \mid \% \\ \mathit{cmp\_op} & ::= & = \mid < \mid > \end{array}$$

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`. On y trouve beaucoup d'autres informations, mais vous pouvez aussi poser des questions.

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`. Testez en exécutant dans le shell, par exemple, `parse 'toto'` ou `parse "x1+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 "0x2a"= 42`.

Question 4. Idem pour les chaînes de caractères (lexème `STRING`). On veut que l'analyse lexicale de `"\"hop\""` donne `STRING("\"hop\"")` et non pas `STRING("\\\"hop\\\"")` [1]. L'affichage de l'expression après parsing devrait être `"\"hop\""`, car il est lui aussi échappé, mais pas `"\\\"hop\\\""`. Pour supprimer les backslashes d'escaping, on pourra utiliser `Scanf.unescaped`.

Question 5. Compléter les règles pour les opérateurs.

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. Ajouter les règles pour `BIN_MULT`, `BIN_CMP`, `AND`, et `OR` dans la grammaire, en suivant le modèle des règles existantes.

Question 9. 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

1. Si vous trouvez que cela fait beaucoup de backslashes, dites vous qu'il y en a encore plus dans mon source TEX. Ou allez lire `https://xkcd.com/1638/`.

cours. Identifiez le(s) conflit(s) associé 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`
Il est aussi possible de déclarer plusieurs tokens au même niveau, par exemple

```
%left TOKEN1 TOKEN2
```

Question 10. 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 11. Faites de même pour `BIN_CMP`, `AND`, `OR` et `NOT`. On utilisera les priorités usuelles, par exemple, on veut que

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

donne

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

Question 12. Il devrait vous rester trois conflits shift/reduce. Analysez `parse.output` pour comprendre d'où ils proviennent. Donnez un exemple d'entrée ambigue pour chacun d'entre eux. Comme précédemment, réglez le problème via des indications de précédences et associativité. Pour les règles réductibles comportant plusieurs tokens, on pourra utiliser la déclaration `%nonassoc` pour le token le plus à droite dans la règle.

Nous nous proposons maintenant d'étendre notre langage avec une notion d'appel de fonction. On utilisera la syntaxe `f(e_1,...,e_N)` où `f` est une variable, les `e_i` sont des expressions, et `N` est un entier positif ou nul. On devra par exemple accepter `toto()`, `tutu(1)`, ou encore `tata(tutu(),toto(1,2))`.

Question 13. Réalisez cette extension :

1. Ajoutez un token pour la virgule.
2. Ajoutez une règle pour produire ce token dans l'analyseur lexical.
3. Ajoutez des règles grammaticales pour supporter l'application de fonction dans l'analyseur syntaxique. (Pour cette étape, le plus simple est de choisir des règles non ambigues.) Pour les productions de la grammaire, on construira des expressions telles que `App("toto",[Int 1, Int 2])`.

Question 14. 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 15. Modifiez l'analyseur syntaxique pour qu'il n'accepte plus que des commandes *cmd* en entrée, et non des expressions *expr*.

Question 16. Bonus. Faites en sorte que l'analyseur syntaxique produise un AST dans lequel les expressions constantes sont déja évaluées. Par exemple, `1+(2*3)+x` devra donner `+(7,x)`.