Analyse Syntaxique — Licence 3

Travaux Pratiques 2014

Auteur : Vincent HUGOT — vhugot@femto-st.fr
Encadrant : Hadrien BRIDE — hadrien.bride@femto-st.fr

7 janvier 2014

Préliminaires techniques

Au cours de ces TPs, nous utiliserons les outils ANLTR et Menhir (≈ ocamlyacc++). On travaillera de préférence sous Linux.

(1) Télécharger ANTLRWorks : `http://www.antlr.org/works`, et tester que cela fonctionne (commande `java -jar antlrworks.jar`).

(2) (Linux, pour TP 2 et suivants) Tester l'installation de Menhir (commande `menhir`) et DOT (commande `dot -V`).

Sur votre ordinateur personnel : Sous Debian et dérivés (*buntu, Mint,...) : `apt-get install graphviz menhir` (en `root`) installera ce qu'il faut pour les TP 2 et suivants.

1 Introduction à l'analyse lexicale avec ANTLRWorks

ANTLR est un générateur de lexers et de parsers LL(*), qui présente quelques fonctions avancées pour la génération des AST et les transformations d'arbres. Il cible Java, C, et d'autres langages. ANTLRWorks est une interface graphique autour d'ANTLR, censée faciliter la prise en main de l'outil —malheureusement elle est affreusement buggée, comme on a pu s'en rendre compte au TP 1.

(1) Créer un fichier `tplexer.g`, et l'ouvrir avec ANTLRWorks.

(2) Écrire `grammar tplexer; LETTER : 'a'..'z'; start: LETTER EOF;` (indenté proprement) et vérifier la grammaire (Ctrl+R).

(3) Se familiariser avec l'interpréteur et le débuggeur. Évaluer `a`, `b`, `A`, `a2`, `2a`.

(4) Constater que ANTLRWorks est buggé (interpréteur et débuggeur, encore que les bugs soient différents). Expliquer la règle `start: LETTER EOF;`, et expliquer en quoi le comportement de l'outil diffère de ce qui est attendu.

(5) Soit la règle suivante :

```
WS : ( ' '| '\t' | '\r' | '\n' ) {$channel=HIDDEN;};
```

Pouvez-vous deviner ce que fait cette règle ?

Donner les règles lexicales suivantes :

(6) `INT` acceptant les nombres entiers (base 10).

(7) `HEX` acceptant les nombres entiers hexadécimaux (e.g. `0x12`, `0X1AF`, `0xff`).

(8) `ID` acceptant les identifiants (e.g. `thing`, `thing2`, `_thing`, `nice2_thing3`, `a23b`,...). On ne peut pas commencer par un chiffre. Commenter la règle `LETTER` pour éviter les conflits...

(9) `FLOAT` acceptant les nombres flottants (e.g. `500.0`, `500.`, `5e2`, `5E2`, `5e+2`, `5E+2`, `0.05`, `.05`, `5e-2`).

(10) `STRING` acceptant les chaînes de caractères, supportant les échappements (e.g. `"blah"`, `""`, `"Blah\nBackslash: \\."`).

Pendant la séance : questions **(1)** à **(5)**.
Rédiger et rendre : questions **(6)** à **(10)**.

2 Introduction à l'analyse syntaxique avec Menhir

Menhir est un générateur de parsers LR(1) ; il améliore et généralise ocamlyacc (LALR(1)), et fait partie de la famille yacc, Bison etc. Comme la plupart des outils de cette famille, il travaille généralement en tandem avec un générateur de lexers de la famille lex, flex etc : en l'occurrence, ocamllex. Menhir cible OCaml, et est extrêmement similaire aux autres outils de la famille yacc, qui visent Java, C, C++, etc, que ce soit du point de vue de la syntaxe ou du fonctionnement. Menhir possède aussi des fonctions avancées, comme les productions paramétrées, que l'on ne verra malheureusement pas ensemble —mais j'invite les curieux à aller consulter le manuel de Menhir [(a)].

(1) Télécharger l'archive `AS-TP-2014-files.zip`, contenant les fichiers suivants (ceux que vous devrez un jour modifier sont en gras — pour ce second TP, vous n'aurez besoin de modifier que le parser et le fichier d'entrée) :

a. `helper.ml` : quelques fonctions utiles générales ou spécifiques au parsing, en particulier pour afficher de jolies représentations graphiques de l'AST (cf. `ast.png`). Ce fichier ne vous concerne pas directement (dans la "vraie vie" il faut s'attendre à devoir implanter du code similaire pour votre langage cible, e.g. Java, C++... mais dans le cadre de ce module, ce code vous est fourni). Les deux déclarations de type suivantes sont utiles pour comprendre la fonction `dot` dans le fichier `arithAST.ml` : **`type`** *lbl* = *string* et **`type`** *dot* = *N* **`of`** *lbl* × *dot list*.

b. **`arithAST.ml`** : c'est ici que se trouve la déclaration de type de l'AST que vous allez produire. La fonction `dot` [(b)] produit une représentation simplifiée de l'AST, de type `dot` (cf. plus haut), qui est utilisée par `helper.ml` pour générer les affichages de l'AST (textuel, en ligne, graphique PNG). Quand vous rajoutez des nœuds à votre AST (le type *t*), il vous suffit d'ajouter la conversion correspondante dans la fonction `dot` (si vous oubliez, le compilateur vous le rappellera).

c. **`arithpar.mly`** : le parser. On y trouve les déclarations des tokens (que le lexer utilise), les règles de grammaire LR(1), et la sémantique associée. Attention à distinguer les tokens (en MAJUSCULES), et les constructeurs de l'AST (Capitalisés). Méfiez vous de la coloration syntaxique, si vous utilisez un éditeur de texte qui la fournit : selon l'éditeur, elle peut ne pas être correcte pour les fichiers `mly` et `mll`.

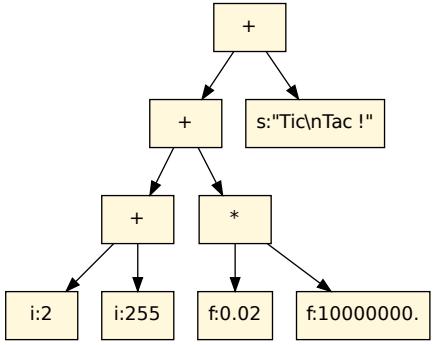
d. **`arithlex.mll`** : le lexer. Pour l'instant, il contient la correction du TP1, ainsi que le support des commentaires —nous y reviendrons ensemble.

e. `arith.ml` : le point d'entrée. Il n'y a pas grand-chose à part une invocation du parser : on analyse le contenu du fichier `arith.in`.

f. `make.sh` : script pour construire et lancer le projet. De nombreuses fonctions de débuggage sont activées, que l'on pourra désactiver une fois que l'on sera plus à l'aise. On utilise le script comme suit : `./make.sh arith`.

g. **`arith.in`** : le fichier d'entrée. Tapez les expressions à analyser dedans, et lancez le projet.

h. `ast.png` : un exemple d'arbre de syntaxe abstraite produit par le programme. Il sera écrasé et remplacé par un nouvel arbre après chaque analyse réussie. En voici un exemple, pour l'entrée `2 + 0xFF + .02 * 1e+7 + "Tic\nTac !"` :



(2) Utiliser `./make.sh arith`, et vérifier que vous obtenez bien, après d'éventuels erreurs/warnings à la compilation (si vous avez modifié les fichiers), une sortie ressemblant à cela :

```
> +
>   +
>     +
>       +
>         i:2
>         i:255
>       f:0.02
```

(a). Manuel etc : `http://gallium.inria.fr/~fpottier/menhir/`. Notons que le manuel de Menhir se concentre sur les nouvelles fonctions par rapport à ocamlyacc. Tutorials sur ocamllex et ocamlyacc :
`http://plus.kaist.ac.kr/~shoh/ocaml/ocamllex-ocamlyacc/ocamllex-tutorial`
`http://plus.kaist.ac.kr/~shoh/ocaml/ocamllex-ocamlyacc/ocamlyacc-tutorial`.
Notons que ces documents décrivent en détail le fonctionnement général de n'importe-quels générateurs de lexers/parsers des familles lex/yacc, à quelques ε près relatifs au langage cible. Ils sont en fait basés sur les manuels de flex et Bison, respectivement.

(b). DOT est un outil de la suite GraphViz qui permet de générer facilement des représentations graphiques de graphes, d'arbres etc. J'utilise cet outil pour que vous ayez une représentation visuelle de l'AST, d'où le nom de ce type.

```
>     f:10000000.
>   s:\"Tic\\nTac !\"
((((i:2 + i:255) + f:0.02) + f:10000000.) + s:\"Tic\\nTac !\")
```

En plus de la sortie console, `ast.png` est généré, pour un total de trois représentations différentes de l'AST. Les fichiers suivants sont générés à la compilation (dans le dossier `_build`) et sont utiles pour résoudre les conflits s'il y a des problèmes dans votre grammaire :

a. `arithpar.automaton` décrit l'automate LR(1) généré, dont voici un état :

```
State 6:
arith -> arith . PLUS arith [ TIMES PLUS EOF ]
arith -> arith . TIMES arith [ TIMES PLUS EOF ]
start -> arith . EOF [ # ]
-- On TIMES shift to state 7
-- On PLUS shift to state 9
-- On EOF shift to state 11
```

b. `arithpar.conflicts` : si votre grammaire est problématique, vous pourrez voir des messages de ce type à la compilation :

```
Warning: 2 states have shift/reduce conflicts.
Warning: 3 shift/reduce conflicts were arbitrarily resolved.
```

Dans ce cas, un fichier `.conflicts` est généré, qui explique l'origine des conflits. Ce fichier fait référence aux états de l'automate généré.

De plus, si la compilation est réussie, le fichier suivant est généré à l'exécution dans le dossier courant :

c. `parser-actions.log` : indique toutes les actions effectuées par le parser :

```
Lookahead token is now INT (0-1)
State 0:
Shifting (INT) to state 3
State 3:
...
Reducing production start -> arith EOF
State 5:
Accepting
```

(3) Examiner le lexer, et comprendre la correction des expressions régulières demandées lors du TP1.

(4) Que pouvez-vous dire a priori des précédences respectives des opérateurs + et *, telles qu'elles sont usuellement définies ? Déterminer expérimentalement (en observant l'AST construit par le parser pour certaines entrées judicieusement choisies) si le parser respecte ces précédences.

(5) De même, que pouvez-vous dire a priori de l'associativité des opérateurs + et * ? Déterminer expérimentalement (en observant l'AST) comment l'associativité de ces opérateurs est gérée par le parser.

(6) Supprimer les lignes `%left PLUS` et `%left TIMES`, et reprendre les deux dernières questions.

(7) Rajouter ces deux lignes, mais en les intervertissant, et reprendre ces mêmes questions.

(8) Ces déclarations sont très pratiques, mais n'existent généralement pas dans les outils LL, il faut donc savoir s'en passer. Écrire une production `int_plus_left`, qui reconnaisse des expressions de la forme `1+2+3+4+...` en utilisant une associativité à gauche, *sans utiliser d'annotation* `%left` *ou* `%right`. Pour tester, on pourra modifier la production `start` en `start: int_plus_left EOF { $1 }`. (On ignorera les flottants et les autres opérateurs, c'est l'associativité qui nous intéresse ici).

(9) Même question pour la production `int_plus_right`, avec une associativité à droite, cette fois.

(10) À votre avis, y a-t-il une relation entre certaines propriétés des productions et l'associativité des opérateurs décrits par ces productions ? Si oui, quelle est cette relation, exactement ? Est-ce important pour tous les opérateurs ? Est-ce important pour certains opérateurs ? Examples ? Cela ne vous inspire-t-il pas une question évidente concernant les générateurs de parsers LL ?

Pendant la séance : questions **(1)** à **(7)**.
Rédiger et rendre : questions **(4)** à **(10)**.

3 Précédences manuelles et plein d'opérateurs...

(1) *(difficile sans avoir vu l'algo LR en détail, garder pour TP3)* Pour un opérateur associatif, quelle est l'implémentation (et donc l'associativité) la plus efficace pour le parsing LR ? Contraster avec le parsing LL.

(2) Combinons maintenant les questions d'associativité et de précédence. Donner des productions telles que `manual_arith_plus_r` reconnaisse des expressions mélangeant sommes et produits d'entiers (e.g. `1+2+3*7*9+4+5`), avec une associativité à droite pour les opérateurs somme et produit, en respectant les priorités usuelles.

(3) Même question `manual_arith_plus_l` avec des associativités gauches.

(4) Une fois que l'on aura bien compris ces notions, on pourra se permettre d'utiliser les annotations `%left` et `%right`. Mettons-les à profit... En utilisant ces annotations si besoin, étendre le lexer et la production `arith` de manière à couvrir les quatre opérateurs binaires usuels de l'arithmétique : `+ * / -`. Vous pouvez commenter les productions `int_plus_left`, `int_plus_right`, `manual_arith_plus_r`, `manual_arith_plus_l`, et leurs sous-productions, on ne s'en servira plus —mais retenez l'idée, que l'on utilisera avec ANTLR dans quelques séances.

(5) Quel niveau de précédence attend-on de l'opposé (moins unaire) ? Implantez le support de l'opposé dans `arith`. Déterminez expérimentalement comment sa précédence est gérée par le parser. Ce problème peut-il être résolu uniquement à l'aide d'annotations `%left` et `%right` ? Une fois que vous avez compris le problème, utilisez la solution temporaire qui consiste à noter l'opposé `~`.

La vraie solution est soit de modifier la grammaire manuellement, soit d'utiliser un token "virtuel", appelons-le `UMINUS`*, donné avec la bonne précédence par l'annotation* `%left UMINUS` *(l'associativité indiquée n'intervient en fait pas en l'occurrence), et d'assigner sa précédence à la règle décrivant la négation unaire au moyen d'une annotation* `%prec UMINUS`*. On aura donc une production du genre* `arith: MINUS arith { ... } %prec UMINUS`.

(6) Étendre la grammaire pour supporter les expressions et opérateurs logiques, comparaisons numériques etc. Opérateurs à supporter : `&&`, `||`, `~` (négation logique ¬), `==`, `<=`, `>=`, `<`, `>`, `<>`. Assurez-vous qu'il n'y ait pas de conflits lecture-réduction, et que les précédences soient cohérentes.

(7) À votre avis, les parsers des langages de programmation usuels acceptent-ils des expressions comme `2 && 3` ? Quelle phase du processus de compilation cela concerne-t-il ?

Pendant la séance : Correction TP2, questions **(1)**, **(4)**, **(5)** (partiel).
Rédiger et rendre : questions **(2)**, **(3)**, **(5)**, **(6)**, **(7)**.

4 Constantes, affectations & instructions...

(1) Ajouter le cas des constantes booléennes `true` et `false`. On utilisera les nœuds `True` et `False` dans l'AST. Lignes à ajouter dans la fonction `dot` :

```
| False → Dot.N ("#f", []) (*false is rendered #f in the visual output*)
| True  → Dot.N ("#t", []) (*true  is rendered #t in the visual output*)
```

(2) Ajouter le cas de l'accès aux éléments d'un tableaux, e.g. `tab_var[i+2]`. On utilisera le nœud `Index of` *string* $\times$ *t* dans l'AST [(c)]. Ligne à ajouter dans la fonction `dot` :

```
| Index (id,x) → Dot.N (id^"[.]", [dot x])
```

(3) Ajouter le cas des expressions parenthésées. Il va sans dire que les parenthèses doivent avoir priorité sur les précédences et associativités des opérateurs. On n'ajoutera évidemment pas de nœud à l'AST, le seul rôle des parenthèses étant de dicter la construction de l'AST.
e.g. `1+2*3+4` et `1+(2*3)+4` donnent l'arbre `(+ (+ 1 (* 2 3)) 4)`.
e.g. `(1+2)*(3+4)` doit donner l'arbre `(* (+ 1 2) (+ 3 4))`.

(4) On veut maintenant gérer une affectation. Utiliser un nouveau non-terminal pour les instructions (par exemple `stmt` pour "statement"), qui remplacera `expr` dans l'axiome `start`. On gardera le même type pour l'AST, en ajoutant simplement un nœud `Assign of` $t \times t$. Je recommande également d'utiliser un non-terminal `assignable` pour désigner les entités auxquelles on peut assigner une valeur, i.e. les identifiants et les éléments d'un tableau. Code à ajouter à `dot` :

```
| Assign (l,r) → Dot.N (":=", [dot l; dot r])
```

e.g. `TRex = 42+i;` ou encore `diploducus[i+1] = (1+2)*23;`.

(5) *Théorique.* Revenons aux questions de précédence et d'associativités. Donner l'algorithme qui, étant donné n operateurs binaires $\star_1, \ldots, \star_n$, d'associativités $a_1, \ldots, a_n \in \{l, r\}$ (a_i étant l'associativité de $\star_i$, l signifiant "gauche" et r "droite"), et de précédences croissantes ($\star_1 < \star_2 < \cdots < \star_n$, par exemple si $n = 2$ on pourrait avoir $\star_1 = +$ et $\star_2 = \times$), génère les productions acceptant les expressions utilisant ces opérateurs, en respectant priorités et associativités dans la construction de l'AST —les atomes (entiers, flottants,...) seront simplement notés α. Par exemple pour $+$ et $\times$, gauches, on devra générer quelque-chose d'équivalent à

$$x^+ \to x^\times \mid x^+ + x^\times \qquad\qquad x^\times \to \alpha \mid x^\times \times \alpha\,.$$

On utilisera du pseudo-code clair et concis pour exprimer l'algorithme.

Rédiger et rendre : questions **(1)** à **(5)** — à rendre plus tard, avec le TP 5.

(c). On pourrait utiliser `Index of` $t \times t$ pour permettre des accès plus complexes, par exemple `tab[k][i]` mais on va faire plus simple pour l'instant.

5 Blocs d'instructions, opérateurs postfixés, boucles. . .

(1) On veut gérer une liste d'instructions (en l'occurrence, des affectations) délimitées par l'éternel point-virgule final. Utiliser un nouveau non-terminal `stmts` et un nouveau nœud `Stmts of t list` dans l'AST. Voici des règles suggérées pour construire le nœud : on utilise `stmts_inner` pour construire la liste d'instructions (une liste d'AST), et `stmts` se contente d'"enrober" le résultat dans un nœud `Stmts` de l'AST. C'est plus pratique que d'essayer de réaliser cet "enrobage" au fur et à mesure que l'on construit la liste. (`SEMI` est mon terminal pour `";"`)

```
stmts: l=stmts_inner { Stmts l }
stmts_inner:(*epsilon*){ [] } | s=stmt SEMI ss=stmts_inner { s::ss }
```

Notez que c'est juste une suggestion, il vous faudra peut-être modifier cette règle plus tard pour mieux gérer les points virgules, selon la façon dont vous abordez les questions suivantes. Code à ajouter à `dot` :

```
  | Stmts l → Dot.N ("<stmts>", map dot l)
```

e.g. `TRex = 42+i; diplodocus[i+1] = (1+2)*23;`

(2) Ajouter les opérateurs `+=`, `-=`, `++` et `--` (postfix, affectations ne renvoyant pas de valeur). On pourrait ajouter de nouveaux nœuds à l'AST, mais dans ce cas précis, je vous demande de "tricher" et d'utiliser les nœuds existants.
e.g. `x=1; x+=2; x-=3; x--; x++;`

Indication : pour gérer l'opposé au dernier TP, on aurait pu simplement renvoyer `Bin (Minus, Int 0, expr)` au lieu d'ajouter `Un` et `UMinus` et de renvoyer `Un (UMinus, expr)`. C'est déjà de la réécriture d'arbre, en "vrai" on évite généralement de faire ça au niveau du parser, mais c'est un bon exercice. . .

(3) Un bloc d'instructions (entre `{ }`) doit également être une instruction. Arrangez-vous pour ne pas avoir besoin de terminer les blocs par des point-virgules —attention : selon la façon dont vous avez écrit votre grammaire aux questions précédentes, c'est plus ou moins difficile. N'hésitez pas à tout commenter et repartir à zéro. Une façon de faire est d'utiliser un non-terminal `terminated_stmt` pour distinguer les instructions qui ont besoin d'un point-virgules de celles qui n'en ont pas besoin —ça facilitera l'implantation des boucles `for` et `while` plus tard.
e.g. `x=2; {y=5; z=6;} t=3;`

(4) Gérer les boucles `while`. Dans l'exemple ci-dessous, `xxx=2;` ne doit pas faire partie de la boucle. Les parenthèses autour de la condition sont optionnelles, contrairement à C. L'instruction `while` n'est pas terminée par un point-virgule, similairement à C.
e.g. `while (x >= 0 && x <= len-1) {x++; tab[x] = x*x;}  xxx=2;`
e.g. `while x<4 x++;` (le point-virgule est lié à `x++`, pas à `while`).

Rédiger et rendre : questions **(1)** à **(4)**, sans oublier le TP 4.

6 Plus de boucles, conditionnelles, opérateur ternaire.

(1) Gérer les boucles `for`, suivant les mêmes principes que pour la question précédente. Notez que les points-virgules et les parenthèses dans `(i = 0; i > len; i++)` font partie de la syntaxe de `for`.
e.g. `for (i = 0; i > len; i++) {thing=i; tab[i]=x;} xxx=2;`

(2) Gérer les boucles `do while`. Notons que, contrairement à `while` et `for`, ces instructions sont délimitées par un point-virgule final. Les parenthèses autour des conditions sont optionnelles.
e.g. `do {x++; tab[x] = x*x;} while (x >= 0 && x <= len-1);  xxx=2;`
e.g. `do x++; while x-2;`

(3) Gérer les instructions conditionnelles de la forme `if else`, le `else` étant obligatoire. Les parenthèses autour des conditions sont optionnelles.
e.g. `if (x >= 0 && x <= len-1) {x++; tab[x] = x*x;} else x--; xxx=2;`
e.g. `if cond if inner x=1; else x=2; else x=3;`

(4) Gérer l'opérateur conditionnel ternaire `?:`. Par exemple, les instructions suivantes sont équivalentes (mais n'ont pas le même AST, évidemment), il faudra donc veiller à assigner les bonnes précédence et associativité au nouvel opérateur :
e.g. `if a < b x=0-2; else x=1+5;` et `x = a<b? 0-2 : 1+5;`
e.g. `if c x=1; else if d x=2; else x=3;` et `x = c? 1 : d? 2 : 3;`

(5) Ajouter une règle pour gérer également le cas où le `else` est omis après `if`. Dans ce cas, la sémantique implicite du cas "sinon" est de ne rien faire, ce qui correspond à l'AST `Stmts []`. Quel conflit obtenez-vous, que le premier exemple ci-dessous illustre ? Quel choix Menhir (ou tout autre outil) fait-il dans ce cas ?
e.g. `if c1 if c2 s=1; else s=2;`
e.g. `if c1 {if c2 s=1; else s=2;}`
e.g. `if c1 {if c2 s=1;} else s=2;`

Rédiger et rendre : questions **(1)** à **(5)**