TP numéro 3 : (Ocaml)lex et (Ocaml)yacc

Assembleur – Compilation, ENSIIE

Semestre 3, 2023–24

Quand on cherche à analyser un fichier de programme pour en construire l'arbre de syntaxe abstraite, on distingue en général deux phases : l'analyse lexicale permet de découper le fichier en mots, appelés tokens, qui servent ensuite d'éléments de base pour l'analyse syntaxique proprement dite.

Lex et yacc, et leur version pour OCaml ocamllex et ocamlyacc, sont des outils qui permettent de générer des analyseurs lexicaux et syntaxiques à partir respectivement d'expressions régulières et de grammaires hors contexte. On trouvera à l'adresse https://ocaml.org/manual/lexyacc.html une documentation complète des outils ocamllex et ocamlyacc que nous allons utiliser au cours de ce TD.

Un certain nombre de fichiers vous est fourni à l'adresse http://www.ensiie.fr/~guillaume.burel/compilation/TP3.tar.gz .

1 Ocamllex

Un fichier ocamllex est de la forme

```
{
    entête
}
rule nom_de_la_fonction = parse
  | regexp_1 { action_1 }
  | ⋮
  | regexp_n { action_n }
{
    trailer
}
```

où l'entête et le trailer (optionnels) peuvent contenir n'importe quel code OCaml qui sera recopié dans le fichier généré. Les expressions régulières suivent le format décrit dans la fiche qui vous a été distribuée. Les actions peuvent être n'importe quel code OCaml qui est exécuté quand une expression régulière a été reconnue.

1. Écrire un fichier `lexer.mll` avec une fonction que vous appelerez `decoupe` dont les expressions régulières reconnaissent :
   - le caractère `+` ;
   - le caractère `*` ;

— la parenthèse ouvrante ;
— la parenthèse fermante ;
— le retour à la ligne ;
— les séquences d'espaces et de tabulations ;
— les constantes entières positives ;
— les identifiants composé d'une lettre (majuscule ou minuscule) ou d'un underscore puis d'une suite de longueur quelconque (éventuellement nulle) de lettres, de chiffres ou d'underscore.

Pour les actions, on affichera un message décrivant ce qui vient d'être reconnu, par exemple `{ Printf.printf "Plus" }` ou
`{ Printf.printf "Identifiant␣%s" (Lexing.lexeme lexbuf) }`

2. Compiler le fichier à l'aide d'ocamllex :
   `ocamllex lexer.mll`
3. Par curiosité, regarder le contenu du fichier généré `lexer.ml`. Ne pas le modifier ! En faisant
   `ocamlc -i lexer.ml`
   regarder le type de la fonction `decoupe`.
   `Lexing.lexbuf` est le type des flux de caractères qui sont utilisés dans ocamllex et ocamlyacc, ils peuvent être créés à partir de fichiers, de chaînes de caractères, etc.
4. À l'aide du `Makefile` et du fichier `test_decoupe.ml` fournis, produire l'exécutable `test_decoupe`. Le fichier `test_decoupe.ml` crée un `Lexing.lexbuf` sur l'entrée standard (à l'aide de `Lexing.from_channel stdin`) ; puis appelle en boucle la fonction `decoupe` du module `Lexer` que vous avez généré.
5. Tester.

2 Ocamlyacc

Un fichier ocamlyacc est de la forme

```
%{
  entête
%}
%token TOKEN_1 ... TOKEN_n
%token <type> TOKEN_m
%start non-terminal_de_depart
%type <type_de_retour> non-terminal_de_depart
%%
grammaire à action
```

Une grammaire avec actions

$$A \to w_1\ \{a_1\} \mid \ldots \mid w_n\ \{a_n\}$$

est écrite avec la syntaxe

a : w_1 { a_1 }
| ⋮
| w_n { a_n }

6. Écrire un fichier `parser.mly` décrivant la grammaire des expressions arithmétiques :

$$S \to E \text{ fin\_de\_ligne}$$
$$E \to E + E \mid E \times E \mid (E) \mid C(n) \mid Id(s)$$

(en mettant des actions vides pour le moment).

7. Compiler avec `ocamlyacc`. Quel message s'affiche-t-il ?
8. Pour supprimer les conflits, outre la technique vue en cours et TD d'ILSF pour désambiguër la grammaire, il est possible avant la ligne `%%` d'indiquer des priorités et des sens d'association aux tokens, par exemple :
```
%left PLUS
%right TIMES
```
rend `PLUS` associatif à gauche et moins prioritaire que `TIMES` qui associe à droite. Essayer avec la grammaire de la question 6, recompiler avec `ocamlyacc`.
9. Par curiosité, regarder le contenu du fichier généré `parser.ml`. Ne pas le modifier ! Regarder le type de la fonction `s` dans le fichier généré `parser.mli`.
10. Le fichier fourni `ast.ml` définit un type pour les arbres de syntaxe abstraite des expressions arithmétiques, et une fonction pour les afficher. Modifier les actions de `parser.mly` pour qu'elles produisent un arbre de syntaxe abstraite, par exemple `{ Plus($1, $3) }`. Il pourra être opportun de rajouter `open Ast` dans l'entête, ne pas oublier de modifier le type de retour de `s`.

3 Combinaison

11. Dans `parser.mli` on constatera qu'un type `token` a été défini. Modifier `lexer.mll` pour que les actions retournent le `token` correspondant à l'expression régulière, par exemple `{ ID (Lexing.lexeme lexbuf) }`. Il pourra être opportun de mettre `open Parser` dans l'entête.
12. Utiliser le `Makefile` et le fichier `test_parser.ml` fournis pour produire l'exécutable `test_parser` qui lit l'entrée standard, la découpe en `token` à l'aide de `decoupe`, la transforme en arbre de syntaxe abstraite avec `s`, puis affiche cet arbre de syntaxe abstraite.
13. Tester.

4 Ajout de constructions syntaxique et analyse sémantique

On veut rajouter des `let` dans nos expressions. On aimerait donc pouvoir écrire des expressions de la forme `let` x `=` e_1 `in` e_2 où x est un identifiant et e_1 et e_2 sont des expressions arithmétiques (qui peuvent donc également contenir des `let`).

14. Modifier `lexer.mll` pour pouvoir reconnaître le signe `=` et les mots clefs `let` et `in`. Attention ! Pour que ces derniers ne soient pas reconnus comme des identifiants, il faudra placer les expressions régulières qui les reconnaissent avant celle reconnaissant les identifiants.
15. Modifier `ast.ml` et `ast.mli` pour ajouter un constructeur `Let` au type `ast`. On sera également amené à ajouter un cas dans la fonction `aff_aux`.
16. Modifier `parser.mly` pour ajouter la déclaration des tokens pour `=`, `let` et `in` ; et pour ajouter une règle permettant de reconnaître la construction `let ... in`.
17. Dans `test_parser.ml`, ajouter une fonction `check_scope` qui prend un AST en entrée et qui renvoie un booléen qui sera vrai ssi la portée des identifiants est correcte dans l'expression, c'est-à-dire que si un identifiant `x` apparaît, il doit être dans la sous-expression e_2 d'un `let x =` e_1 `in` e_2.
18. Tester.

On veut maintenant rajouter la possibilité d'avoir des fonctions anonymes, comme en OCaml. On veut donc pouvoir écrire des expressions de la forme `fun` x `->` e où x est un identifiant et e une expression.

19. Modifier `lexer.mll` pour pouvoir reconnaître le signe `->` et le mot clef `fun`.
20. Modifier `ast.ml` et `ast.mli` pour ajouter un constructeur `Fun` au type `ast`. On sera également amené à ajouter un cas dans la fonction `aff_aux`.
21. Modifier `parser.mly` pour ajouter la déclaration des tokens pour `->` et `fun` ; et pour ajouter une règle permettant de reconnaître la construction `fun ... -> ...`.
22. Dans `test_parser.ml`, modifier la fonction `check_scope` pour prendre en compte les fonctions anonymes : dans `fun` x `->` e, x est bien dans la portée de e.
23. Tester.
24. Modifier la grammaire pour pouvoir reconnaître des expressions de la forme `fun` x_1 x_2 `...` x_n `->` e. On ne changera pas la syntaxe abstaite, car une telle expression sera reconnue de la même façon que `fun` x_1 `-> fun` x_2 `-> ... fun` x_n `->` e.
25. Modifier la grammaire pour pouvoir reconnaître des expressions de la forme `let` f x_1 x_2 `...` x_n `=` e_1 `in` e_2. On ne changera pas la syntaxe abstaite, car une telle expression sera reconnue de la même façon que `let` f `= fun` x_1 `-> fun` x_2 `-> ... fun` x_n `->` e_1 `in` e_2.

On veut maintenant ajouter la possibilité d'appliquer une expression à une autre, comme dans un langage fonctionnel. On pourra ainsi reconnaître des expressions comme `f 2 (3 + 1)`.

26. Modifier `ast.ml` et `ast.mli` pour ajouter un constructeur `App of ast * ast` au type `ast`. On sera également amené à ajouter un cas dans la fonction `aff_aux`.

27. Modifier `parser.mly` pour que la grammaire puisse reconnaître les applications.

    Indication 1 : les applications doivent être parenthésées à gauche. Ainsi, `f 1 x` doit être compris comme `(f 1) x`.

    Indication 2 : on pourra grouper les expressions en trois niveaux de priorité : les constructions `let ... in`, `fun` et les opérateurs arithmétiques qui sont de priorité moindre, les applications de priorité moyenne, et les constantes, variables et expressions entre parenthèses de plus haute priorité. On construira la grammaire correspondante, sans chercher à utiliser les mécanismes d'ocamlyacc.