# Analyse syntaxique

Julien Forget
julien.forget@onera.fr
www.cert.fr/anglais/deri/jforget/ML.html

30 avril 2009

# Plan

# Analyse syntaxique : définition et structuration

- Analyse syntaxique : découpage d'un texte brut en termes complexes (en programme). Analyse de la forme d'un texte.

$\neq$ Analyse sémantique : analyse du sens d'un texte (ex : typage).

- Décomposition de l'analyse syntaxique :
  1. Analyse lexicale : décomposition d'un texte en différents mots (ex : en Français découpage selon les espaces).
  2. Analyse grammaticale : regroupement des mots en différentes constructions (ex : en Français, découpage en propositions).

$\Rightarrow$ Production d'une représentation abstraite du texte d'entrée : l'Arbre de Syntaxe Abstraite.

# Exemple

## Example

```
let x = 4 in
x+3
```

- L'analyse lexicale reconnaît les symbôles suivants :

# Exemple

## Example

```
let x = 4 in
x+3
```

- L'analyse lexicale reconnaît les symbôles suivants : **let**, **in** (mots-clés), x (identifiant), 4, 3 (entiers), =, + (caractères spéciaux).
- L'analyse grammaticale regroupe les mots clés en expressions, puis les expressions "simples" en expressions plus complexes. Par exemple

# Exemple

## Example

```
let x = 4 in
x+3
```

- L'analyse lexicale reconnaît les symbôles suivants : **let**, **in** (mots-clés), x (identifiant), 4, 3 (entiers), =, + (caractères spéciaux).
- L'analyse grammaticale regroupe les mots clés en expressions, puis les expressions "simples" en expressions plus complexes. Par exemple elle reconnaît que x+3 est une *application de fonction* et que **let** x = 4 définit une *liaison*.
- Par contre cette analyse ne permet ni de reconnaître que x est de type entier ni de savoir que l'expression complète a pour valeur 7.

# Plan

# Analyse lexicale

- L'analyse lexicale consiste à décomposer un flux de caractères d'entrée en une suite d'éléments plus abstraits : les lexèmes (token).
- Il s'agit en général du premier traitement effectué par un compilateur.
- Exemple de lexèmes : nombres, chaînes de caractères.
- Un lexème est en général défini par une expression régulière.

# Les expressions régulières en Unix

- De nombreux outils Unix manipulent les expressions régulières (shell, sed, awk).
- Par exemple, le shell :
  - `?` : caractère quelconque (une fois).
  - `*` : caractère quelconque, répétitions quelconque.

## Noms de fichiers

```
% ls *.ml
% ls toto.ml?
```

# (Caml)lex

- *lex* est un outil d'analyse lexicale Unix.
- Il prend en entrée un ensemble d'expressions régulières (et les actions associées) et produit en sortie un automate reconnaissant le langage correspondant.
- *camllex* est une version OCaml de lex.
- `ocamllex` *lexer*`.mll` : produit l'automate OCaml correspondant dans *lexer*`.ml`.

# Forme générale

## Code

```
{header}
rule entrypoint = parse
  regexp1 {action1}
| ...
| regexpn {actionn}
{trailer}
```

- *regexp*1, $\cdots$, *regexpn* : des expressions régulières.
- *action*1, $\cdots$, *actionn* : des expressions OCaml décrivant l'action à réaliser lorsqu'une expression régulière est reconnue.
- {*header*} et {*trailer*} : un code OCaml quelconque. Sert en général à définir des fonctions auxilliaires, des déclarations préliminaires, etc.

# Syntaxe des expressions régulières Camllex

- "*string*" : une chaîne de caractère ;
- '*char*' : un caractère ;
- ['$char_1$' '$char_2$' ...] : *char*1 ou $char_2$ ou ... ;
- ['$char_1$' - '$char_2$'] : un caractère compris entre $char_1$ et $char_2$ ;
- eof : la fin du fichier ;
- *regexp** : *regexp* répétée un nombre quelconque de fois ;
- *regexp*+ : *regexp* répétée au moins une fois ;
- _ : n'importe quelle chaîne de caractères ;
- etc.

# Expressions régulières : exercice

- Définissez le lexème entier.
- Définissez le lexème identifiant.

# Les actions

- Une action consiste à construire la valeur correspondant à un lexème.
- On utilise pour cela des constructeurs (comme pour les types union).
- On peut récupérer le contenu du lexème reconnu grâce à l'expression Lexing.lexeme lexbuf (a pour type string).

**Example**

```
['0'-'9']+ {INT (int_of_string
                   (Lexing.lexeme lexbuf))}
```

# Exemple : lexer d'expressions arithmétiques

## lexer.mll

```
{open Parser;;}

rule token = parse
| '+' {PLUS}
| '-' {MOINS}
| '*' {MULT}
| '/' {DIV}
| ['0'-'9']+ {INT (int_of_string
                    (Lexing.lexeme lexbuf))}
| ['A'-'z']+ {IDENT (Lexing.lexeme lexbuf)}
```

# Plan

# Grammaires : rappel

Une grammaire est un quadruplet $G =< V, T, P, S >$, avec :

- $V$ l'ensemble des variables.
- $T$ l'ensemble des terminaux.
- $P$ l'ensemble des règles de production.

$$P \subseteq (V \cup T)^* V (V \cup T)^* \times (V \cup T)^*$$

- $S \in V$ le symbole de départ.

**Example**

$$\begin{aligned} V &= S \\ T &= a, b, c \\ P &= S \rightarrow aSbS, S \rightarrow c \end{aligned}$$

Mots reconnus :

# Grammaires : rappel

Une grammaire est un quadruplet $G =< V, T, P, S >$, avec :

- $V$ l'ensemble des variables.
- $T$ l'ensemble des terminaux.
- $P$ l'ensemble des règles de production.

$$P \subseteq (V \cup T)^* V (V \cup T)^* \times (V \cup T)^*$$

- $S \in V$ le symbole de départ.

**Example**

$$\begin{aligned} V &= S \\ T &= a, b, c \\ P &= S \rightarrow aSbS, S \rightarrow c \end{aligned}$$

Mots reconnus : $c, acbc, aacbcbc, ...$

# Grammaires (2)

- Dans une grammaire *non-contextuelle*, toutes les règles de $P$ sont de la forme : $V_i \longrightarrow w$ où $V_i \in V$ et $w \in (V \cup T)^+$
- Arbre de dérivation d'une grammaire :
  - Racine : $S$.
  - Nœud : une variable.
  - Feuille : un terminal.
- Mot $w$ vérifié par une grammaire $\Leftrightarrow$ il existe un arbre dont les feuilles lues dans l'ordre forment $w$.
- Dans une grammaire ambiguë, plusieurs arbres de dérivation peuvent aboutir au même mot.

**Grammaire non-contextuelle mais ambiguë**

$P = S \longrightarrow aSSb, S \longrightarrow aSbS, S \longrightarrow b$

Le mot *abbb* correspond à plusieurs arbres de dérivation.

# Arbre de syntaxe abstraite

- Un arbre de syntaxe abstraite (AST) permet de représenter une construction d'un langage de programmation.
- Dans un AST, un nœud correspond à un opérateur dont les fils sont les opérandes.
- Les feuilles correspondent soit à une constante soit à une variable de la grammaire du langage.
- L'analyse syntaxique consiste à construire l'AST correspondant au programme pris en entrée. C'est la première phase de la compilation.
- Les phases suivantes de la compilation (moderne) consistent à effectuer une suite d'analyses et de transformations sur cet arbre.

# (Caml)yacc

- *yacc* est un outil d'analyse grammaticale Unix.
- Il prend en entrée un ensemble de règles de grammaires et produit un analyseur syntaxique reconnaissant le langage correspondant.
- Les grammaires acceptées sont non-contextuelles mais pas forcément non-ambiguës (signalé par des warnings).
- Tout comme pour lex, *ocamlyacc* produit un fichier `.ml` contenant l'analyseur syntaxique (à partir d'un fichier `.mly`).

# Forme générale

## Code

```
%{ header %}
 declarations
%%
 rules
%%
 trailer
```

# Déclaration des lexèmes

- Les règles de la grammaire yacc manipulent des lexèmes définis dans le lexer associé.
- Les différents lexèmes doivent être déclarés dans le parser à l'aide du mot clé *token* :
  - %token constr1 ... constrn : rajoute les lexèmes *constr*1, ... *constrn*.
  - %token <typexpr> constr1 constrn : fait de même mais attache en plus un attribut de type *typexpr* aux lexèmes.
- Règles d'associativité et de priorité entre opérateurs :
  - % left constr1 constr2 : ces opérateurs sont associatifs à gauche (% right pour droite, %nonassoc pour non associatif).
  - Lorsqu'on a plusieurs lignes avec de telles déclaration, les opérateurs les plus prioritaires sont ceux apparaissant en dernier.

# Exemple : parser d'expressions arithmétiques

## parser.mly (1/2)

```
%{ open Ast;; %} (* Header *)
(* Lexemes et leur type *)
%token <int> INT
%token <string> IDENT
%token PLUS MOINS MULT DIV

(* Regles d'associativite et priorites *)
%left PLUS MOINS
%left MULT DIV

(* Symbole de depart et son type *)
%start exp
%type <Ast.expr> exp
%%
```

# Les règles de grammaire

## Forme Générale

```
nonterminal :
    symbol ... symbol {action}
  | ...
  | symbol ... symbol {action}
```

- *symbol* correspond ici soit à un lexème soit à un autre non-terminal (une autre règle de la grammaire).
- *action* construit la valeur correspondant à la (sous)-règle reconnue.
- Dans une action, $1 permet d'accéder à la valeur produite par l'action associée au premier symbole de la règle, $2 au deuxième, etc.

# Exemple : parser d'expressions arithmétiques (2)

## parser.mly (2/2)

```
(* Les regles (une seule ici) *)
exp :
  INT { ExpInt $1 }
| IDENT { ExpVar $1 }
| exp PLUS exp { ExpBin ($1, Plus, $3) }
| exp MOINS exp { ExpBin ($1, Moins, $3) }
| exp MULT exp { ExpBin ($1, Mult, $3) }
| exp DIV exp { ExpBin ($1, Div, $3) }
```

# Plan

# Programme principal

Il faut maintenant :

- Définir les structures de données (l'AST) construites par le parser.
- Invoquer le parser et le lexer sur un fichier d'entrée.
- Compiler le tout.

# Exemple : AST

ast.ml

```
type binop = Plus | Moins | Mult | Div

type expr =
    Expint of int
  | ExpVar of string
  | ExpBin of expr * binop * expr
```

# Programme principal

arith.ml

```
open AST;;

let string_of_expr e =
  match e with
    | ExpInt i -> string_of_int i
    | ExpVar v -> v
    | ...

let _ =
  let lexbuf = Lexing.from_channel
      (open_in "fic.txt") in
  let e = Parser.expr Lexer.token lexbuf in
  Printf.fprintf (string_of_expr e)
```

# Compilation

```
ocamllex lexer.mll       # -> lexer.ml
ocamlyacc parser.mly     # -> parser.ml et parser.mli
ocamlc -c parser.mli
ocamlc -c lexer.ml
ocamlc -c parser.ml
ocamlc -c ast.ml
ocamlc -c arith.ml
ocamlc -o arith lexer.cmo parser.cmo ast.cmo arith.cmo
```