Programmation Dirigée par la Syntaxe (PDS)

CM3 - Analyse syntaxique
de la syntaxe concrète à la syntaxe abstraite

ISTIC, Université de Rennes 1
Sebastien.Ferre@irisa.fr

PDS, M1 info

Plan

1. Introduction
2. Concepts
   - Grammaires et arbres de dérivation
   - Analyse syntaxique
   - Grammaires attribuées
3. Mise en oeuvre
   - Implémentation
   - Grammaires étendues
4. Exemples

Plan

1 Introduction

2 Concepts

3 Mise en oeuvre

4 Exemples

Introduction

- La syntaxe abstraite
  - facilite la manipulation par programme
  - explicite les structures de phrases
  - est représentée par des arbres
- La syntaxe concrète
  - est plus facile à lire et écrire par des humains
  - est plus proche de la langue naturelle
    mots outils, ponctuation
  - est représentée par des séquences
    de caractères, de mots, de sons, ...

Problèmes

- Comment définir une syntaxe concrète ?
- Comment passer de la syntaxe concrète (séquences) à de la syntaxe abstraite (arbres)
  - c'est-à-dire, comment retrouver la structure cachée dans la séquence ?

Plan

1 Introduction

2 Concepts
- Grammaires et arbres de dérivation
- Analyse syntaxique
- Grammaires attribuées

3 Mise en oeuvre

4 Exemples

Définition d'une syntaxe concrète

Analogie

- ASD : définition d'une syntaxe abstraite
- grammaire : définition d'une syntaxe concrète
  - il existe plusieurs familles de grammaires formant la hiérarchie de Chomsky
  - ici, les grammaires hors-contexte
    - LA famille pour les langages informatiques
    - imbrications arbitraires de structures syntaxiques

Définition d'une syntaxe concrète

Une syntaxe concrète est généralement décomposée en 2 niveaux :

1. Lexical : des caractères aux mots
   - caractères : ASCII ou Unicode
   - mots : unités lexicales (anglais, *token*)
     ex : identificateurs, nombres, mots-clés, ...
   - définition par expressions régulières
2. Syntaxique : des mots aux phrases
   - mots : ceux définis au niveau lexical
   - phrases : expressions, programmes, requêtes, ...
   - définition par des grammaires hors-contexte

Plan

1 Introduction

2 Concepts
- Grammaires et arbres de dérivation
- Analyse syntaxique
- Grammaires attribuées

3 Mise en oeuvre

4 Exemples

Grammaire

Definition

Une grammaire est composée :

- d'un ensemble de règles syntaxiques
  - chaque règle associe à un symbole non-terminal un ensemble de productions
  - chaque production est une séquence de symboles
  - chaque symbole est soit un non-terminal, soit une unité lexicale, soit une chaîne de caractères
  - un non-terminal joue le rôle de "racine" (appelé *axiome*)
- d'un ensemble de règles lexicales
  - chaque règle associe à une unité lexicale une expression régulière

Exemple : grammaire des expressions

(1)……

Parallèle entre grammaire et ASD

grammaire	ASD
symbole non-terminal	type défini
unité lexicale	type de base
mots-clés, ponctuation	constructeur
règle	définition de type
production	variant

Langage engendré

Le langage défini par une grammaire *G* est :

- le langage engendré par *G*
- l'ensemble des phrases engendrées par *G*

Une phrase est engendrée par *G* si elle est obtenue

- en partant de l'axiome de la grammaire
- en appliquant une chaine de dérivations
- jusqu'à avoir une séquence de caractères
  càd. plus de dérivation possible

Une dérivation consiste à

- remplacer un non-terminal par une de ses productions
- remplacer une unité lexicale par une séquence de caractères "matchant" l'expression régulière la définissant

Langage engendré

Le langage défini par une grammaire *G* est :

- le langage engendré par *G*
- l'ensemble des phrases engendrées par *G*

Une phrase est engendrée par *G* si elle est obtenue

- en partant de l'axiome de la grammaire
- en appliquant une chaine de dérivations
- jusqu'à avoir une séquence de caractères
  càd. plus de dérivation possible

Une dérivation consiste à

- remplacer un non-terminal par une de ses productions
- remplacer une unité lexicale par une séquence de caractères "matchant" l'expression régulière la définissant

Langage engendré

Le langage défini par une grammaire *G* est :

- le langage engendré par *G*
- l'ensemble des phrases engendrées par *G*

Une phrase est engendrée par *G* si elle est obtenue

- en partant de l'axiome de la grammaire
- en appliquant une chaine de dérivations
- jusqu'à avoir une séquence de caractères
  càd. plus de dérivation possible

Une dérivation consiste à

- remplacer un non-terminal par une de ses productions
- remplacer une unité lexicale par une séquence de caractères "matchant" l'expression régulière la définissant

Exemple de dérivation : `x+y*z`

(2)……

Arbre de dérivation (*parse tree*)

Un arbre de dérivation

- est la trace d'une chaine de dérivations
- dans laquelle on a "oublié" l'ordre des dérivations élémentaires

C'est un arbre :

- noeud x :
  - une occurence de dérivation pour une production $x \rightarrow \alpha$
  - chaque α_i est la racine d'un sous-arbre ou une feuille
- feuille A :
  - une unité lexicale

Arbre de dérivation (*parse tree*)

Un arbre de dérivation

- est la trace d'une chaine de dérivations
- dans laquelle on a "oublié" l'ordre des dérivations élémentaires

C'est un arbre :

- noeud x :
  - une occurence de dérivation pour une production $x \rightarrow \alpha$
  - chaque α_i est la racine d'un sous-arbre ou une feuille
- feuille A :
  - une unité lexicale

Exemple d'arbre de dérivation : x+y*z

(3)……

Exemple d'arbre de dérivation : `(x+y)*z`

(4)……

Grammaire ambigüe

Definition

Une grammaire est ambigüe si il existe plusieurs arbres de dérivation pour une même phrase, càd. plusieurs façons de l'engendrer.

Exemple : (5). . . . . .

Rmq : c'est au concepteur de la grammaire de veiller à éviter toute ambiguité.

Plan

1 Introduction

2 Concepts
- Grammaires et arbres de dérivation
- Analyse syntaxique
- Grammaires attribuées

3 Mise en oeuvre

4 Exemples

Analyse syntaxique

- si on voit une grammaire comme un type
  - alors ses vraies valeurs sont les arbres de dérivation
  - et les phrases en sont des formes dégénérées
- un problème important est donc de retrouver la structure perdue dans une phrase
- c'est ce qu'on appelle l'analyse syntaxique
- seulement bien définie si la grammaire n'est pas ambigüe

Analyse syntaxique : théorie et pratique

- C'est un domaine très mature
  aussi bien en théorie qu'en pratique
- Théorie : langages formels
  - analyse descendante OU ascendante OU tabulée
  - automates à pile
  - automate des items non-contextuels
- Pratique : compilateurs d'analyseurs syntaxiques efficaces
  - entrée : grammaire (règles syntaxiques et lexicales)
  - sortie : analyseur syntaxique
    fonction : phrase $\rightarrow$ arbre de dérivation
  - outils : Lex&Yacc, JavaCC, ANTLR, ...

Analyse syntaxique : principe

En 2 phases :

- Analyse lexicale
  - découpage de la séquence de caractère en séquence d'unités lexicales
  - compilation des expressions régulières en automates finis
  - chaque automate essaye de reconnaitre la prochaine unité
- Analyse syntaxique : approche descendante
  - on lit la phrase de gauche à droite
  - on maintient une pile de symbole
  - si le symbole en sommet de pile est :
    - une unité lexicale : on consomme la prochaine unité dans la phrase
      erreur si différentes
    - un non-terminal x : on le remplace dans la pile par une production de x en fonction de la prochaine unité dans la phrase et on développe un noeud de l'arbre de dérivation
  - succès si la pile et le mot sont vides

Exemple d'analyse descendante : `x+y*z`

(6)……

Grammaire LL(k)

- La clé de l'efficacité tient dans le déterminisme de l'analyse
- En analyse descendante, l'indétermination est dans le choix de la production remplaçant le sommet de pile
- Solution : grammaires LL(k)
  - LL = *Left-to-right scan, Leftmost derivation*
  - la production peut être choisie en lisant au plus *k* unités lexicales
  - LL(1) : il suffit de lire une unité lexicale
- Certaines grammaires non-LL(k) peuvent être transformées pour le devenir
  - sans changer le langage engendré, bien sûr !
  - mais cela change les arbres de dérivation, qui deviennent souvent plus complexes

Grammaire LL(1) pour les expressions

(7)……

Nouvel arbre de dérivation pour `x+y*z`

(8)……

Plan

1 Introduction

2 Concepts
- Grammaires et arbres de dérivation
- Analyse syntaxique
- Grammaires attribuées

3 Mise en oeuvre

4 Exemples

Grammaires attribuées

- Une ASD attribuée permet de définir un calcul guidé par une structure arborescente
- Le même type de calcul peut être fait sur un arbre de dérivation
  - propagation de valeurs
  - synthétisé : des feuilles vers la racine
  - hérité : de la racine vers les feuilles
- Une grammaire attribuée
  - définit un calcul dirigé par la syntaxe concrète
  - en décorant la grammaire avec
    - des attributs (hérités et synthétisés)
    - des équations définissant les attributs entre eux

Fonction des grammaires atribuées

- En principe, toutes les fonctions peuvent être définies par grammaire attribuée
  - pretty-printing
  - évaluation
  - compilation
  - ...
- Mais ces fonctions sont plus simples à définir sur la syntaxe abstraite
  - RAPPEL propriétés : précision, abstraction, simplicité
- On se limite ici à la fonction
  - phrase $\rightarrow$ AST
  - soit, l'inverse du *pretty-printing*

Grammaires attribuées : définition

D'après l'analogie grammaire / ASD

Definition

Une grammaire attribuée est une grammaire où :

- chaque non-terminal x est décoré par des attributs $x.a, x.b, \ldots$
- chaque production est décorée par des calculs sur ces attributs
  - définissant certains attributs en fonction des autres

- les attributs et calculs sont de même nature
- les contraintes sur les dépendances sont les mêmes
  + pas de propagation de droite à gauche
- mais l'attribut *self* n'est pas défini
  l'arbre de dérivation n'a pas besoin d'être construit

Grammaires attribuées : définition

D'après l'analogie grammaire / ASD

Definition

Une grammaire attribuée est une grammaire où :

- chaque non-terminal x est décoré par des attributs $x.a, x.b, \ldots$
- chaque production est décorée par des calculs sur ces attributs
  - définissant certains attributs en fonction des autres

- les attributs et calculs sont de même nature
- les contraintes sur les dépendances sont les mêmes
  + pas de propagation de droite à gauche
- mais l'attribut *self* n'est pas défini
  l'arbre de dérivation n'a pas besoin d'être construit

Exemple : Expressions : attributs

(9)……

Exemple : Expressions : calculs (1/2)

$$
\begin{array}{lcl}
\mathit{expr} & \rightarrow & \mathit{fact}\ \mathit{exprAux} \\
 & & \left\{ \begin{array}{lcl} \mathit{exprAux.left} & := & \mathit{fact.ast} \\ \mathit{expr.ast} & := & \mathit{exprAux.ast} \end{array} \right. \\
\mathit{exprAux} & \rightarrow & '+'\ \mathit{fact}\ \mathit{exprAux}_1 \\
 & & \left\{ \begin{array}{l} \mathit{exprAux}_1\mathit{.left} := \\ \quad \textbf{Binop}(\textbf{Plus}, \mathit{exprAux.left}, \mathit{fact.ast}) \\ \mathit{exprAux.ast} := \mathit{exprAux}_1\mathit{.ast} \end{array} \right. \\
 & | & '-'\ \mathit{fact}\ \mathit{exprAux}_1 \\
 & & \left\{ \begin{array}{l} \mathit{exprAux}_1\mathit{.left} := \\ \quad \textbf{Binop}(\textbf{Minus}, \mathit{exprAux.left}, \mathit{fact.ast}) \\ \mathit{exprAux.ast} := \mathit{exprAux}_1\mathit{.ast} \end{array} \right. \\
 & | & \varepsilon \\
 & & \{\ \mathit{exprAux.ast} := \mathit{exprAux.left}
\end{array}
$$

Exemple : Expressions : calculs (2/2)

$$
\begin{array}{lcl}
\mathit{fact} & \rightarrow & \mathit{term}\ \mathit{factAux} \\
 & & \left\{ \begin{array}{lcl} \mathit{factAux.left} & := & \mathit{term.ast} \\ \mathit{fact.ast} & := & \mathit{factAux.ast} \end{array} \right. \\
\mathit{factAux} & \rightarrow & '*'\ \mathit{term}\ \mathit{factAux}_1 \\
 & & \left\{ \begin{array}{l} \mathit{factAux}_1\mathit{.left} := \\ \quad \textbf{Binop}(\textbf{Times}, \mathit{factAux.left}, \mathit{term.ast}) \\ \mathit{factAux.ast} := \mathit{factAux}_1\mathit{.ast} \end{array} \right. \\
 & | & \varepsilon \\
 & & \{\ \mathit{factAux.ast} := \mathit{factAux.left} \\
\mathit{term} & \rightarrow & \mathit{INT} \\
 & & \{\ \mathit{term.ast} := \textbf{Const}(\mathit{INT.value}) \\
 & | & \mathit{ID} \\
 & & \{\ \mathit{term.ast} := \textbf{Var}(\mathit{ID.name}) \\
 & | & '('\ \mathit{expr}\ ')' \\
 & & \{\ \mathit{term.ast} := \mathit{expr.ast}
\end{array}
$$

Exemple : Expressions : AST de `x+y*z`

(11). . . . . .

Plan

1 Introduction

2 Concepts

3 Mise en oeuvre
- Implémentation
- Grammaires étendues

4 Exemples

Plan

1 Introduction

2 Concepts

3 Mise en oeuvre
- Implémentation
- Grammaires étendues

4 Exemples

Implémentation

1. OCaml : *stream parsers* pour grammaires $LL(1)$
2. Java : ANTLR pour grammaires $LL(*)$

Implémentation OCaml

- *stream parsers*
  - type `'a stream` : séquence de valeurs de type `'a`
  - *pattern matching* sur les *streams*
    - *pattern* = production
    - unité lexicale *A* : `'A` (un quote devant)
    - non-terminal *x* : `s1, s2 = x h1 h2`
      ⇒ arguments = attributs hérités (`h1,h2`)
      ⇒ résultats = attributs synthétisés (`s1,s2`)
- analyse lexicale : `char stream` → `token stream` où `token` est le type des unités lexicales
- analyse syntaxique : `token stream` → τ où τ est un type d'AST

Exemple : Expressions : type des unités lexicales

```
type token =
  | PLUS | MINUS | TIMES
  | LEFT | RIGHT
  | INT of int
  | ID of string
```

Exemple : Expressions : analyseur syntaxique

```
module Parser = struct
  let rec expr = parser
    | [< f=fact; e=expr_aux f >] -> e
  and expr_aux left = parser
    | [< ' PLUS; f=fact; e=expr_aux (Binop (Plus,left,f)) >] -> e
    | [< ' MINUS; f=fact; e=expr_aux (Binop (Minus,left,f)) >] -> e
    | [< >] -> left
  and rec fact = parser
    | [< t=term; f=fact_aux t >] -> f
  and fact_aux left = parser
    | [< ' TIMES; t=term; f=fact_aux (Binop (Times,left,t)) >] -> f
    | [< >] -> left
  and term = parser
    | [< ' INT i >] -> Const i
    | [< ' ID s >] -> Var s
    | [< ' LEFT; e=expr; ' RIGHT >] -> e
end
```

Exemple : Expressions : analyseur lexical

```
let rec lexer = parser
  | [< ' (' ' | '\n' | '\t'); toks=lexer >] -> toks
  | [< tok=token; toks=lexer >] -> [< 'tok; toks >]
  | [< >] -> [< >]
and token = parser
  | [< ' '+' >] -> PLUS
  | [< ' '-' >] -> MINUS
  | [< ' '*' >] -> TIMES
  | [< ' '(' >] -> LEFT
  | [< ' ')' >] -> RIGHT
  | [< '('0'..'9' as c); i = int (Char.code c - Char.code '0') >]
                -> INT i
  | [< '('a'..'z' as c); s = id (String.make 1 c)>]
                -> ID s
and int acc = parser
  | [< ' ('0'..'9' as c);
       i=int (10*acc + (Char.code c - Char.code '0')) >] -> i
  | [< >] -> acc
and id acc = parser
  | [< ' ('a'..'z'|'A'..'Z'|'0'..'9'|'_' as c);
       s=id (acc ^ String.make 1 c) >] -> s
```

Exemple : Expressions : analyseur lexical

Dans un fichier `lexer.mll`

```
let digit = ['0'-'9']
let letter = ['A'-'Z' 'a'-'z']
let blank = [' ' '\t' '\n']

rule tokenize = parse
  | '+'   { [< ' PLUS; tokenize lexbuf >] }
  | '-'   { [< ' MINUS; tokenize lexbuf >] }
  | '*'   { [< ' TIMES; tokenize lexbuf >] }
  | '('   { [< ' LEFT; tokenize lexbuf >] }
  | ')'   { [ ' RIGHT; tokenize lexbuf >] }
  | (digit+ as lxm)   { [< ' INT (string_of_int lxm); tokenize lexbuf >
  | (letter+ as lxm)   { [< ' ID lxm; tokenize lexbuf >] }
  | blank   { tokenize lexbuf }
  | _ as c   { failwith ("unexpected character: " ^ String.make 1 c) }
```

Exemple : Expressions : analyse syntaxique

```
let channel = open_in filename in
let lexbuf = Lexing.from_channel channel in
let tokens = Lexer.tokenize lexbuf in
let ast = Parser.expr tokens in
...
```

Rmq : Compiler le code OCaml avec l'option `-pp camlp4o` pour inclure l'extension syntaxique des *stream parsers*.

Implémentation Java avec ANTLR

- ANTLR = ANother Tool for Language Recognition
- C'est un compilateur d'analyseur syntaxique
  - entrée : grammaire attribuée
  - sortie : code Java de l'analyseur syntaxique + calculs
  - peut aussi générer du C++, C#, Python, Javascript, ...
- Composition fichier ANTLR : extension .g (grammaire)
  1. options : langage cible, ...
  2. règles syntaxiques
  3. règles lexicales
- Commande de compilation de l'analyseur syntaxique
  - `java org.antlr.Tool Expr.g`
  - génère les fichiers `ExprLexer.java` et `ExprParser.java`

Notations ANTLR

- non-terminaux : en minuscules `expr`
- unités lexicales : en majuscules `INT`, OU entre *quotes* `'+'`
- attributs : paramètres et résultats des non-terminaux
  s=*x*`[`*h*1`,` *h*2`]`
- référence à un attribut :
  - `$`*a* pour le non-terminal en tête de règle
  - `$`*x*`.`*a* pour les symboles *x* de la production
- calculs en langage cible entre accolades `{ ... }`
  - n'importe où dans une production
  - `@init { ... }` : action initiale commune aux productions

Exemple : Expressions : grammaire ANTLR (1/2)

```
grammar Expr;

options {
  language=Java; // target language for generated parser
}

// syntactic rules
expr returns [Expr ast]
    : f=fact e=expr_aux[ $f.ast ] { $ast = $e.ast; }
    ;
expr_aux [Expr left] returns [Expr ast]
    : '+' f=fact e=expr_aux[ new Binop(new Plus(), $left.ast, $f.ast)
        { $ast = $e.ast; }
    | '-' f=fact e=expr_aux[ new Binop(new Minus(), $left.ast, $f.ast)
        { $ast = $e.ast; }
    |
        { $ast = $left; }
    ;
fact returns [Expr ast]
    : t=term f=fact_aux[ $t.ast ] { $ast = $f.ast; }
    ;
[...]
```

Exemple : Expressions : grammaire ANTLR (2/2)

```
fact_aux [Expr left] returns [Expr ast]
    : '*' t=term f=fact_aux[ new Binop(new Times(), $left, $t.ast)]
        { $ast = $f.ast; }
    |
        { $ast = $left; }
    ;
term returns [Expr ast]
    @init { system.out.println("Parsing a term..."); }
    : INT  { $ast = new Const(Integer.parseInt($INT.text)); }
    | ID   { $ast = new Var($ID.text); }
    | '(' e=expr ')'  { $ast = $e.ast; }
    ;

// ignoring whitespaces
WS  : (' '|'\n'|'\t')+  { $channel = HIDDEN; } ;
// lexical rules
INT : ('0'..'9')+ ;
ID  : ('a'..'z') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
// $
```

Exemple : Expressions : analyse syntaxique

```
try {
    // reading the file
    ANTLRFileStream input = new ANTLRFileStream(filename);
    // creating the stream of tokens as lexer output
    ExprLexer lexer = new ExprLexer(input);
    CommonTokenStream tokenStream = new CommonTokenStream(lexer);
    // calling the parser
    ExprParser parser = new ExprParser(tokenStream);
    Expr ast = parser.expr();
    ...

} catch (RecognitionException re) {
    re.printStackTrace();
}
```

Plan

1 Introduction

2 Concepts

3 Mise en oeuvre
- Implémentation
- Grammaires étendues

4 Exemples

Grammaires étendues

- Même opérateurs d'extension que pour les ASD
  - optionnel : ?
  - multiple : $*$, $+$
- Les productions
  - ne sont pas limitées à des séquences
  - mais sont des expressions régulières
  - y compris les alternatives (|)
- Cela permet des grammaires plus concises
- Les actions peuvent être placées dans la portée de ces opérateurs d'extension

Exemple : expressions

(12). . . . . .

Implémentation dans ANTLR

ANTLR offre un support direct pour les grammaires étendues

```
// syntactic rules
expr returns [Expr ast]
    : f=fact { $ast = $f.ast; }
      ( '+' f2=fact { $ast = new Binop(new Plus(), $ast, $f2.ast); }
      | '-' f2=fact { $ast = new Binop(new Minus(), $ast, $f2.ast); }
    ;
fact returns [Expr ast]
    : t=term { $ast = $t.ast; }
      ( '*' t2=term { $ast = new Binop(new Times(), $ast, $t2.ast); }
    ;
term returns [Expr ast]
    : INT  { $ast = new Const(Integer.parseInt($INT.text)); }
    : ID   { $ast = new Var($ID.text); }
    : '(' e=expr ')'  { $ast = $e.ast; }
    ;
// $
```

Implémentation en OCaml

En OCaml, les opérateurs d'extension peuvent être définis par des fonction d'ordre supérieur

- càd. des fonctions de type *parser* $\rightarrow$ *parser*

```
let option (sub_parser : 'a parser) : 'a option parser =
  parser
  | [< x = sub_parser >] -> Some x
  | [< >] -> None

let alt sub_parser_1 sub_parser_2 =
  parser
  | [< x1 = sub_parser1 >] -> x1
  | [< x2 = sub_parser2 >] -> x2

let rec star (x0 : 'a) (sub_parser : 'a -> 'a parser) : 'a parser =
  parser
  | [< x1 = sub_parser x0; x2 = star x1 sub_parser >] -> x2
  | [< >] -> x0
```

Implémentation en OCaml : exemple expressions

```
module Parser = struct
  let rec expr = parser
    | [< f=fact;
         e = star f (fun left -> parser
             | [< ' PLUS; f=fact >] -> (Binop (Plus,left,f))
             | [< ' MINUS; f=fact >] -> (Binop (Minus,left,f))) >]
       -> e
  and rec fact = parser
    | [< t=term;
         f = star t (fun left -> parser
             | [< ' TIMES; t=term >] -> (Binop (Times,left,t))) >]
       -> f
  and term = parser
    | [< ' INT i >] -> Const i
    | [< ' ID s >] -> Var s
    | [< ' LEFT; e=expr; ' RIGHT >] -> e
end
```

Plan

1 Introduction

2 Concepts

3 Mise en oeuvre

4 Exemples

Exemples

Définition d'une syntaxe concrète pour nos exemples de syntaxe abstraite
grammaires sans règles lexicales et non-attribuées (laissé en exercice !)

1. Expressions régulières
2. Programmes impératifs
3. Programmes fonctionnels
4. Grammaire des grammaires

Exemple : Expressions régulières

$$
\begin{array}{lcl}
\textit{regexp} & \rightarrow & \textit{regexpAlt} \\
\textit{regexpAlt} & \rightarrow & \textit{regexpConcat}\ (\text{'|'}\ \textit{regexpConcat})* \\
\textit{regexpConcat} & \rightarrow & (\textit{regexpClosure})* \\
\textit{regexpClosure} & \rightarrow & \textit{regexpAtom}\ (\text{'*'}|\text{'+'}|\textit{interval})? \\
\textit{interval} & \rightarrow & \text{'\{'}\ \textit{INT}?\ \text{','}\ \textit{INT}?\ \text{'\}'} \\
\textit{regexpAtom} & \rightarrow & \textit{CHAR} \\
 & | & \text{'['}\ \textit{CHAR}+\ \text{']'} \\
 & | & \epsilon \\
 & | & \text{'('}\ \textit{regexpAlt}\ \text{')'}
\end{array}
$$

Exemple : Programmes impératifs (1/2)

Syntaxe concrète à la C

$$
\begin{array}{lcl}
\textit{program} & \rightarrow & \textit{function}* \\
\textit{function} & \rightarrow & (\textit{type} \mid \text{'}\mathbf{void}\text{'})\ \textit{IDENT}\ \text{'('}\ \textit{params}?\ \text{')'}\ \textit{statement} \\
\textit{params} & \rightarrow & \textit{param}\ (\text{','}\ \textit{param})* \\
\textit{param} & \rightarrow & \textit{type}\ \textit{IDENT} \\
\textit{type} & \rightarrow & \text{'}\mathbf{bool}\text{'} \mid \text{'}\mathbf{int}\text{'} \mid \text{'}\mathbf{float}\text{'} \mid \text{'}\mathbf{string} \\
 & \mid & \textit{type}\ \text{'}*\text{'} \\
\textit{statement} & \rightarrow & \textit{place}\ \text{'='}\ \textit{expression}\ \text{';'} \\
 & \mid & \textit{IDENT}\ \text{'('}\ \textit{arguments}\ \text{')'}\ \text{';'} \\
 & \mid & \text{'}\mathbf{return}\ \textit{expression}?\ \text{';'} \\
 & \mid & \text{'\{'}\ \textit{declaration}*\ \ \textit{statement}*\ \ \text{'\}'} \\
 & \mid & \text{'}\mathbf{if}\text{'}\ \text{'('}\ \textit{expression}\ \text{')'}\ \textit{statement}\ (\text{'}\mathbf{else}\text{'}\ \textit{statement})? \\
 & \mid & \text{'}\mathbf{while}\text{'}\ \text{'('}\ \textit{expression}\ \text{')'}\ \textit{statement} \\
\textit{declaration} & \rightarrow & \textit{type}\ \textit{IDENT}\ (\text{'='}\ \textit{expression})?\ \text{';'} \\
\textit{place} & \rightarrow & \textit{IDENT} \mid \textit{place}\ \text{'*'}
\end{array}
$$

Exemple : Programmes impératifs (2/2)

$$
\begin{array}{lcl}
\textit{expression} & \rightarrow & \textit{exprOr} \\
\textit{exprOr} & \rightarrow & \textit{exprAnd}\ (\text{'||'}\ \textit{exprAnd})* \\
\textit{exprAnd} & \rightarrow & \textit{exprNot}\ (\text{'\&\&'}\ \textit{exprNot})* \\
\textit{exprNot} & \rightarrow & \text{'!'}?\ \textit{exprComp} \\
\textit{exprComp} & \rightarrow & \textit{exprPlus}\ (\text{'='}|\text{'!='}|\text{'<'}|\text{'>'}|\text{'<='}|\text{'>='})\ \textit{exprPlus} \\
\textit{exprPlus} & \rightarrow & \textit{exprTimes}\ ((\text{'+'}|\text{'-'})\ \textit{exprTimes})* \\
\textit{exprTimes} & \rightarrow & \textit{exprNeg}\ ((\text{'*'}|\text{'/'}|\text{'\%'})\ \textit{exprNeg}) \\
\textit{exprNeg} & \rightarrow & \text{'-'}?\ \textit{exprAtom} \\
\textit{exprAtom} & \rightarrow & \textit{BOOL} \mid \textit{INT} \mid \textit{FLOAT} \mid \textit{STRING} \\
 & \mid & \text{'NULL'} \\
 & \mid & \textit{IDENT}\ \text{'('}\ \textit{arguments}?\ \text{')'} \\
 & \mid & \text{'('}\ \textit{exprOr}\ \text{')'} \\
\textit{arguments} & \rightarrow & \textit{expression}\ (\text{','}\ \textit{expression})*
\end{array}
$$

Exemple : Programmes fonctionnels

Syntaxe concrète à la Caml

$$
\begin{array}{lcl}
\textit{program} & \rightarrow & \textit{definition}* \\
\textit{definition} & \rightarrow & \text{'}\textbf{let}\text{'}\ \textit{IDENT}\ \text{'='}\ \textit{expression} \\
\textit{expression} & \rightarrow & \textit{exprPair} \\
\textit{exprPair} & \rightarrow & \textit{exprCons}\ \text{','}\ \textit{exprCons} \\
\textit{exprCons} & \rightarrow & \textit{exprApply}\ \text{'::'}\ \textit{exprCons} \\
\textit{exprApply} & \rightarrow & \textit{exprApply}\quad \textit{exprAtom} \\
\textit{exprAtom} & \rightarrow & \textit{BOOL} \mid \textit{INT} \mid \textit{FLOAT} \mid \textit{STRING} \\
 & \mid & \text{'[]'} \\
 & \mid & \textit{IDENT} \\
 & \mid & \text{'}\textbf{function}\text{'}\ \textit{IDENT}\ \text{'->'}\ \textit{exprPair} \\
 & \mid & \text{'('}\ \textit{exprPair}\ \text{')'}
\end{array}
$$

Soyons réflexifs !

La grammaire des grammaires étendues !

grammar	$\rightarrow$	*rule*$*$
rule	$\rightarrow$	*LIDENT* $'\rightarrow'$ *productionAlt*
productionAlt	$\rightarrow$	*productionConcat* $('\|'$ *productionConcat*$)*$
productionConcat	$\rightarrow$	*productionClosure productionClosure*$*$
	$\mid$	$'\epsilon'$
productionClosure	$\rightarrow$	*productionAtom* $('?'\|'*')?$
productionAtom	$\rightarrow$	*STRING* $\mid$ *UIDENT* $\mid$ *LIDENT*
	$\mid$	$'('$ *productionAlt* $')'$

- Attention : ne pas confondre $*$ et $'*'$, $|$ et $'|'$, ...
- unités lexicales : *UIDENT* (noms unités lexicales), *LIDENT* (noms non-terminaux), *STRING* (chaines entre quotes)

The End