Projet du premier semestre L3

Sébastien Briais

ENS Lyon

17 septembre 2007

Rôle d'un compilateur

- Traduire des programmes écrits dans un langage source donné en un langage cible.
- Le langage source : `drei`.
- Le langage cible : assembleur type RISC.

Rôle d'un compilateur

- Traduire des programmes écrits dans un langage source donné en un langage cible.
- Le langage source : `drei`.
- Le langage cible : assembleur type RISC.
- Concrètement... (démo)

Le langage `drei` - 1

- Un mini-langage orientée objet
- Un type primitif : Int
- Instructions de contrôle :
  if (<expression>) <statement> [**else** <statement>]
  while (<expression>) <statement>
- Opérateurs
  - arithmétiques : +, -, *, /, %
  - logiques : ||, &&, !
  - de comparaison : <, <=, >, >=, ==, !=

Le langage `drei` - 2

Calcul de la factorielle

```
{
    var x : Int = 5 ;
    var res : Int = 1 ;
    while (x > 0) {
        set res = x * res ;
        set x = x - 1 ;
    }
    printInt(res) ;
    printChar(10) ;
}
```

Le langage `drei` - 3

Déclarer une classe

```
class Rational {
    val num : Int ;
    val den : Int ;
    def add(that :Rational) : Rational =
        new Rational(
            this.num * that.den + that.num * this.den,
            this.den * that.den) ;
}
```

Déclarer une sous-classe

```
class ExtRational extends Rational {
    def mul(that :Rational) : Rational = ...
}
```

Le langage `drei` - 4

Créer un objet

```
var r : Rational = new Rational(1,3) ;
```

Sélectionner un champ

```
var x : Int = r.num ;
```

Appeler une méthode

```
var s : Rational = r.add(new Rational(1,4)) ;
```

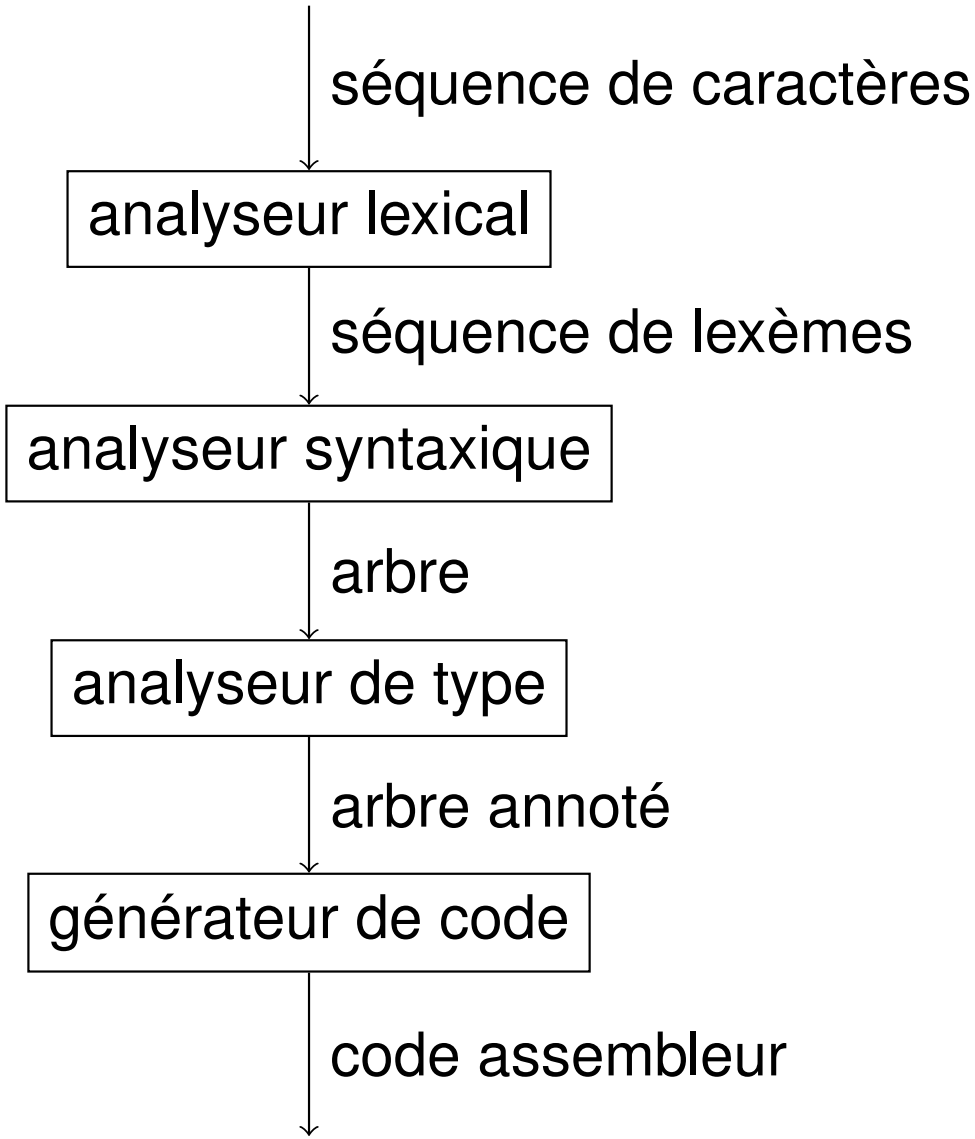
Exercice

- Suite de Fibonacci

$$f_0 = 0$$
$$f_1 = 1$$
$$f_{n+2} = f_n + f_{n+1}$$

- Implémenter une classe List implémentant les listes d'entiers et comprenant les méthodes head, tail, isEmpty et cons.

Structure d'un compilateur



Langages

- Un langage est un ensemble de chaînes de caractères formant les *phrases* du langage.
- La structure des phrases est définie par une *grammaire*.
- Un programme (une phrase du langage) est constitué de mots : les *lexèmes*.
- Les mots sont aussi des chaînes de caractères. La structure d'un mot peut aussi être décrite par une grammaire.

Langages - exemples

Exemple 1

Phrase = Sujet Verbe.
Sujet = “Georges” | “Laetitia”.
Verbe = “cours” | “marche”.

Exemple 2

Nombre = Chiffre | Chiffre Nombre.
Chiffre = “0” | “1” | ... | “9”.

Grammaire

Une grammaire est un quadruplet $G = (V, \Sigma, P, S)$

- V : alphabet non vide des symboles *non terminaux*
- Σ : alphabet non vide des symboles *terminaux*
- $P \subseteq \Gamma^+ \times \Gamma^*$ ensemble fini des *productions*
- $S \in V$ est le *symbole initial*

avec $\Sigma \cap V = \emptyset$ et $\Gamma = \Sigma \cup V$.

Grammaire

Une grammaire est un quadruplet $G = (V, \Sigma, P, S)$

- V : alphabet non vide des symboles *non terminaux*
- Σ : alphabet non vide des symboles *terminaux*
- $P \subseteq \Gamma^+ \times \Gamma^*$ ensemble fini des *productions*
- $S \in V$ est le *symbole initial*

avec $\Sigma \cap V = \emptyset$ et $\Gamma = \Sigma \cup V$.

La grammaire est *non-contextuelle* si toutes les productions sont de la forme $\alpha \rightarrow \beta$ avec $\alpha \in V$.

Dérivation

Soit $G = (V, \Sigma, P, S)$ une grammaire. La relation de *dérivation* est définie pour $\alpha, \beta \in \Gamma^*$ par

$$\alpha \vdash \beta \iff \exists \theta, \theta', \alpha', \beta' \in \Gamma^* \begin{cases} (\alpha', \beta') \in P \\ \alpha = \theta \alpha' \theta' \\ \beta = \theta \beta' \theta' \end{cases}$$

Le langage de G est l'ensemble des mots dérivables depuis le symbole initial, i.e.

$$L(G) = \{u \in \Sigma^* \mid S \vdash^* u\}$$

Dérivation la plus à gauche

La relation de *dérivation la plus à gauche* est définie par

$$\alpha \vdash_{\mathrm{g}} \beta \iff \exists u \in \Sigma^*, \theta, \alpha', \beta' \in \Gamma^* \begin{cases} (\alpha', \beta') \in P \\ \alpha = u\alpha'\theta \\ \beta = u\beta'\theta \end{cases}$$

Si G est une grammaire non contextuelle, on a $L(G) = L_{\mathrm{g}}(G)$ avec

$$L_{\mathrm{g}}(G) = \{u \in \Sigma^* \mid S \vdash_{\mathrm{g}}^* u\}$$

Une grammaire G est *ambiguë* s'il existe un mot dans $L(G)$ qui peut être dérivé de deux façons différentes avec la relation de dérivation la plus à gauche.

Notation BNF

Backus-Naur Form

grammar	=	production grammar | (empty).
production	=	ident "=" expression ".".
expression	=	term | expression "|" term.
term	=	factor | term factor | "(empty)".
factor	=	ident | string.
ident	=	letter | ident letter | ident digit.
string	=	"\"" stringchars "\"".
stringchars	=	stringchars stringchar | (empty).
stringchar	=	escapechar | plainchar.
escapechar	=	"\\" char.
plainchar	=	charNQNE.
char	=	tout caractère imprimable
charNQNE	=	tout caractère imprimable sauf «"» et «\»

Notation BNF étendue (EBNF)

Deux constructions supplémentaires :

- { x } : zéro, une ou plusieurs occurrences de x
- [x] : zéro ou une occurrence de x

Langage régulier

- Un langage est *régulier* si sa syntaxe peut être exprimée à l'aide d'une seule régle EBNF non récursive.
- Les langages réguliers sont reconnaissables par des machines à états finis (automates finis).

Langage régulier

- Un langage est *régulier* si sa syntaxe peut être exprimée à l'aide d'une seule régle EBNF non récursive.
- Les langages réguliers sont reconnaissables par des machines à états finis (automates finis).
- Un langage est régulier si sa syntaxe peut être exprimée par plusieurs règles EBNF qui ne dépendent pas récursivement les unes des autres.

Exemple

identifier = letter { letter | digit }.
digit = "0" | ... | "9".
letter = "a" | ... | "z" | "A" | ... | "Z".

Analyseur lexical

micro- et macro-syntaxe

Dans le contexte des langages de programmation, les mots (ou lexèmes) sont décrit par la *micro-syntaxe* et les phrases (ou programmes) par la *macro-syntaxe*.

analyseur lexical

Il traduit un programme source (séquence de caractères) en une séquence de lexèmes définis par la micro-syntaxe.

Exemple - 1

```
type t
val current_char : t -> char option
val next_char : t -> unit

let is_char c = function
  | Some c' when c = c' -> true
  | _ -> false

let is_in l = function
  | Some c -> List.mem c l
  | None -> false

let is_letter = is_in ['a'; ...; 'z'; 'A'; ...; 'Z']

let is_digit = is_in ['0'; ...; '9']
```

Exemple - 2

```
let ident chars : unit =
  if is_letter (current_char chars) then
    begin
      next_char chars;
      while (is_letter (current_char chars))
          || (is_digit (current_char chars))
        do
          next_char chars
        done
      end
  else failwith "ident"
```

Traduction d'un langage régulier en un programme

Pr("x")

```
assert (is_char 'x' (current_char chars));
next_char chars
```

Pr((exp))

Pr(exp)

Pr([exp])

```
if is_in (first(exp)) (current_char chars)
then Pr(exp);
```

Traduction d'un langage régulier en un programme

Pr({exp})

```
while is_in (first(exp)) (current_char chars) do
  Pr(exp)
done;
```

Pr($fact_1$ ... $fact_n$)

```
Pr(fact_1); ...; Pr(fact_n);
```

Traduction d'un langage régulier en un programme

$Pr(term_1 \mid ... \mid term_n)$

```
if is_in first(term_1) (current_char chars)
then Pr(term_1)
else if is_in first(term_2) (current_char chars)
then Pr(term_2)
...
else if is_in first(term_n) (current_char chars)
then Pr(term_n)
```

Analysable par la gauche

Une grammaire est *analysable par la gauche* si

- $term_1$ | ... | $term_n$: les termes n'ont pas de symboles initiaux en commun (first($term_i$) disjoints deux à deux)
- $fact_1$ ... $fact_n$: si $fact_i$ peut produire le mot vide alors $fact_i$ et $fact_{i+1}$ n'ont pas de symboles initiaux en commun
- [exp], { exp } : l'ensemble des symboles initiaux de exp ne peut pas contenir un symbole qui suit [exp] ou { exp }

Rôle d'un analyseur lexical

Lire une partie de l'entrée et retourner le prochain lexème.

```
type token
type t

val current_token : t -> token
val next_token : t -> unit
```

Élimine les espaces blancs et les commentaires.

Exemple

```
type token =
   | CLASS | ...
   | IDENT of string
   | NUMBER of Int32.t
   |  ...
```

```
class Foo {
  def bar(): Int = 42;
}
```

```
CLASS IDENT("Foo") LBRACE DEF IDENT("bar")
LPAREN RPAREN COLON INT EQUAL NUMBER(42)
SEMICOLON RBRACE
```

Les grammaires non-contextuelles

- Les langages réguliers ne peuvent pas exprimer l'imbrication :
- $\{a^n b^n \mid n \in \mathbb{N}\}$ n'est pas régulier mais algébrique (généré par une grammaire non-contextuelle)

Analyse syntaxique par descente récursive

- Une fonction **let** A chars = ... pour chaque non-terminal *A*
- Les expressions régulières sont traduites comme avant sauf que les non-terminaux sont traduits en un appel à la fonction correspondante.
- Ne marche que pour les grammaires analysables par la gauche.

Exemple

A = “a” A “c” | “b”.

```
let rec A chars =
  if is_char 'a' (current_char chars) then
    begin
      next_char chars;
      A chars;
      assert (is_char 'c' (current_char chars));
      next_char chars
    end
  else
    begin
      assert (is_char 'b' (current_char chars));
      next_char chars
    end
```

Exemple

E = E “+” E | E “*” E | ident | ident “[” E “]” | “(“ E “)”.

Exemple

E = E “+” E | E “*” E | ident | ident “[” E “]” | “(“ E “)”.

Lever l’ambiguïté

E = E “+” T | T.
T = T “*” F | F.
F = ident | ident “[” E “]” | “(“ E “)”.

Exemple

E = E “+” E | E “*” E | ident | ident “[” E “]” | “(“ E “)”.

Lever l’ambiguïté

E = E “+” T | T.
T = T “*” F | F.
F = ident | ident “[” E “]” | “(“ E “)”.

Éliminer la récursivité à gauche

E = T E’.
E’ = “+” T E’ | (empty).
T = F T’.
T’ = “*” F T’ | (empty).
F = ident | ident “[” E “]” | “(“ E “)”.

Exemple

E = E “+” E | E “*” E | ident | ident “[” E “]” | “(“ E “)”.

Factoriser à gauche

F = ident ((empty) | “[” E “]”) | “(“ E “)”.

Exemple

E = E “+” E | E “*” E | ident | ident “[” E “]” | “(“ E “)”.

Factoriser à gauche

F = ident ((empty) | “[” E “]”) | “(“ E “)”.

- Ne marche pas à tous les coups

Un langage LL(2)

S = { A }.
A = ident “ :=” E.
E = { ident }.

Il peut être analysé connaissant deux symboles lus en avance.

Bibliographie

- *Modern Compiler Implementation in ...*, Andrew W. Appel
- *Lex et Yacc*