Théorie des Langages
Épisode 7 — L'analyseur syntaxique

Thomas Pietrzak

Université Paul Verlaine — Metz

Introduction

Le principal inconvénient des méthodes de constructions de table LR est qu'elles exigent de fournir une quantité de travail trop importante pour construire un analyseur LR à la main pour les langages de programmation usuels.

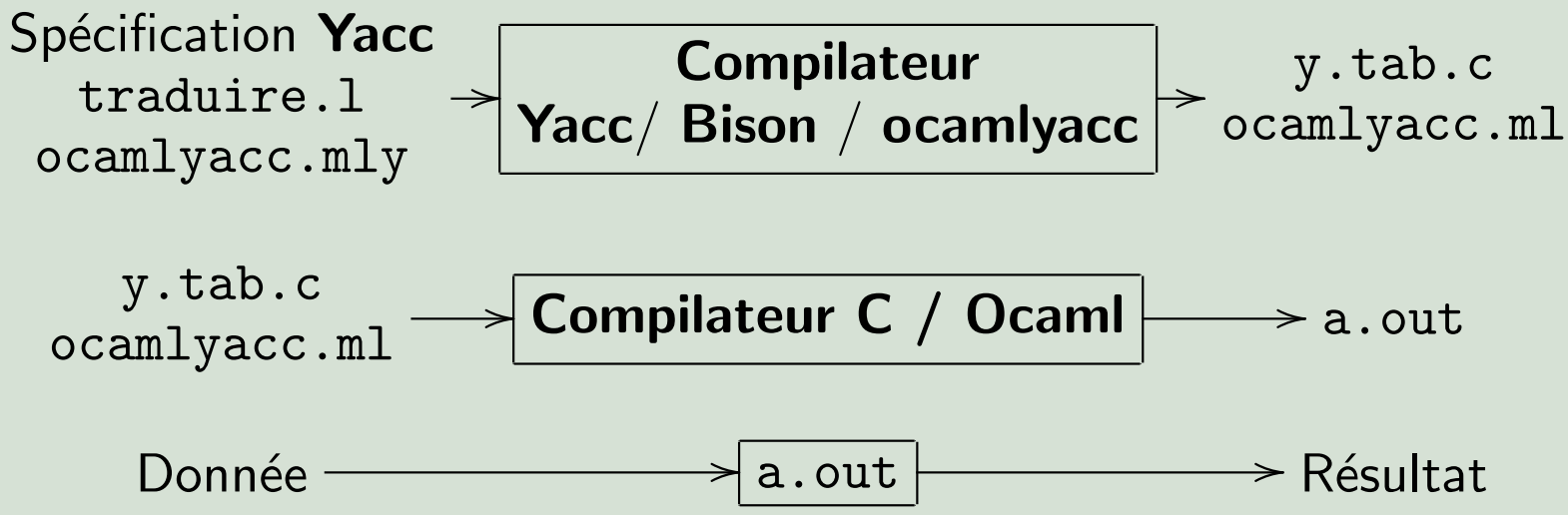
- On a besoin d'un outil spécialisé : un constructeur d'analyseur LR
- Heureusement, de tels constructeurs sont disponibles en grand nombre et nous considérons l'usage de l'un d'entre eux : **Yacc** (OCamlyacc et Camlyacc pour Caml, et Bison en version libre).
- Avec un tel constructeur, on peut écrire une grammaire hors-contexte et la lui soumettre afin qu'il produise automatiquement l'analyseur correspondant.
- Si la grammaire constient certaines ambiguïtés ou d'autres constructions qui sont difficiles à analyser en un parcours gauche-droite du texte source, le constructeur d'analyseurs peut repérer ces constructions et informer le concepteur du compilateur de leur présence.

Introduction

- Nous allons voir comment utiliser un constructeur d'analyseurs pour faciliter la production de la partie frontale d'un compilateur. Nous utiliserons **Yacc** (et ses amis), un constructeur d'analyseur LR.
- **Yacc** est l'acronyme de « Yet another compiler-compiler » (et un compilateur de compilateurs de plus), reflétant la popularité des constructeurs d'analyseurs au début des années 70 quand la première version de Yacc fût conçue par Stephen Curtis Johnson.
- **Yacc** est disponible en tant que commande sur le système UNIX (bison sous linux), et a été utilisé pour faciliter l'implantation de certains compilateurs.

Schéma

On peut construire un traducteur en utilisant **Yacc** de la manière suivante :



Introduction

- On prépare tout d'abord un fichier, par exemple `traduire.y` qui contient une spécification du traducteur pour **Yacc**. La commande système UNIX est :
  `yacc traduire.y` ou `bison traduire.y`
- Cette commande produit un programme C appelé `y.tab.c` utilisant la méthode LR (LALR plus précisément : Look Ahead LR) à partir du fichier `traduire.y`
- Le programme `y.tab.c` est une implémentation d'un analyseur LALR écrite en C, complétée par d'autres routines C que l'utilisateur a éventuellement écrites.
- En compilant `y.tab.c` en utilisant la bibliothèque `ly` (le nom `ly` dépend du système) qui contient le programme d'analyse LR, ce qui se fait par la commande :
  `cc -ly y.tab.c` ou `gcc -ly y.tab.c`
- On obtient `a.out`, le programme objet désiré qui effectue la traduction spécifiée par le programme **Yacc** original.

Spécification en Lex

Un programme **Yacc** consiste en trois parties :

```
déclarations
%%
règles de traduction
%%
routines C annexes
```

En principe, tout ça doit vous rappeler des choses : c'est la même structure qu'un programme Lex.

Partie déclarations

La partie déclarations contient deux parties optionnelles :

- la première contient des déclarations C ordinaires, délimitées par %{ et %}. On inclut ici les déclarations des variables temporaires utilisées par les règles de traduction ou les procédures de la deuxième et troisième section.
- la seconde contient des déclarations d'unités lexicales de la grammaire. Ces unités lexicales peuvent être utilisées dans la deuxième et la troisième partie.

Partie des règles de traduction

Cette partie se situe après le premier %%.

- Dans cette partie on énonce les règles de traduction.
- Chaque règle est formée d'une production de la grammaire et de son action sémantique associée.
- Pour un ensemble de production de type :
  $\langle partie\ gauche \rangle \rightarrow \langle alt_1 \rangle \mid \langle alt_2 \rangle \mid \ldots \mid \langle alt_n \rangle$
  s'écrirait en **Yacc**

```
<partie gauche> : <alt 1> {action sémantique 1}
                  <alt 2> {action sémantique 2}
                  ...
                  <alt n> {action sémantique n}
```

Partie des règles de traduction (suite)

- Dans une production **Yacc**, un caractère simple entre apostrophes comme $'c'$ est considéré comme désignant le symbole terminal c, et les chaînes de lettres et de chiffres non entourées d'apostrophes et non déclarées comme des unités lexicales, sont interprétées comme des non-terminaux.
- Les différentes alternatives d'une même partie gauche peuvent être séparées par une barre verticale (pipe, |).
- Chaque production **Yacc** se termine par un point-virgule (; pour ceux qui auraient pas suivi).
- Les règles de traduction sont donc constituées d'une partie gauche, d'une série d'alternatives et d'actions sémantiques qui leur sont associées.
- La première des parties gauches est considérée comme l'axiome.

PArtie des actions sémantiques

Une action sémantique est une séquence d'instructions en C.

- Dans une action sémantique, le symbole $$ référence la valeur de l'attribut associé au non-terminal de la partie gauche.
- `$i` référence la valeur associée au i^{e} symbole de la grammaire (terminal ou non-terminal) en partie droite.

Partie des routines annexes

La troisième partie des spécifications **Yacc** contiennent des routines écrites en C qui aident à la traduction.

- Un analyseur lexical nommé `yylex()` doit être fourni.
- D'autres procédures comme les routines de récupération d'erreur peuvent être ajoutées si nécessaire.
- L'analyseur lexical `yylex()` produit des couples formés d'une unité lexicale et de la valeur de l'attribut associé.
- Si une unité lexicale telle que `CHIFFRE` est retournée, l'unité lexicale doit être déclarée dans la première section de la spécification **Yacc**.
- La valeur de l'attribut associée à une unité lexicale est communiquée à l'analyseur syntaxique par l'intermédiaire de la variable `yylval` prédéfinie dans **Yacc**.

Exemple

Pour illustrer la préparation d'un programme source en **Yacc**, construisons un calculateur de bureau simplifié qui lit une expression arithmétique, l'évalue et imprime sa valeur numérique. Nous allons le construire en partant de la grammaire des expressions arithmétiques suivante :

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \textit{chiffre}$

L'unité lexicale *chiffre* désigne un chiffre unique compris entre 0 et 9. Le programmer du calculateur de bureau construit par **Yacc** est présenté ci-après...

```
%{
  #include <ctype.h>
%}
%token CHIFFRE

%%

ligne   : expr '\n'          { printf("%d\n",$1); } ;
expr    : expr '+' term      { $$ = $1 + $3; } | term ;
term    : term '*' facteur   { $$ = $1 * $3; } | facteur ;
facteur : '(' expr ')'       { $$ = $2; } | CHIFFRE ;

%%

yylex()
{
  int c;
  c = getchar();
  if (isdigit(c))
  {
    yyval = c - '0';
    return CHIFFRE;
  }
  return c;
}
```

Exemple : explication

- Dans la première partie des déclarations on inclut `ctype.h`, ce qui provoque l'inclusion de ce fichier lors de la compilation de l'analyseur syntaxique. Ce fichier est utile dans notre cas pour avoir la fonction `isdigit`.
- Dans la deuxième partie des déclarations on déclare l'unité lexicale `CHIFFRE`.
- Dans les actions sémantiques on définit les actions associées aux productions de la grammaire. Par exemple pour $E \rightarrow E + T \mid T$ on écrit le code **Yacc** suivant :

```
expr    : expr '+' term     { $$ = $1 + $3; }
        | term;
```

  Notons que le non-terminal `term` dans la première production est le troisième symbole en partie droite tandis que `'+'` est le second.
- L'action sémantique associée à la première production ajoute la valeur de `expr` et la valeur de `term` et affecte le résultat à l'attribut associé au non-terminal `expr` en partie gauche.

Exemple : explication (suite)

- Nous avons omis l'action sémantique associée à la seconde production, car la simple copie de valeur est l'action par défaut sur les productions comportant un unique symbole en partie droite. En général, `{ $$ = $1; }` est l'action sémantique par défaut.
- Remarquons que nous avons ajouté une nouvelle production de départ à la spécification **Yacc** :
  ```
  ligne   : expr '\n'   { printf("%d\n", $1);}
  ```
- Cette production spécifie qu'une chaîne proposée au calculateur de bureau doit être une expression suivie d'un caractère de fin de ligne.
- L'analyseur lexical implémenté dans l'exemple est très simple : il lit les caractères d'entrée un par un. Si le caractère est un chiffre, la valeur de ce chiffre est rangée dans la variable `yylval` et l'unité lexicale `CHIFFRE` est retournée. Sinon le caractère lui-même est retourné en tant qu'unité lexicale.

Utilisation conjointe de Lex et de Yacc

Lex a été conçu pour produire des analyseurs lexicaux qui peuvent être utilisés avec des analyseurs syntaxiques faits avec **Yacc**.

- La bibliothèque **Lex** `ll` fournit un programme moniteur appelé `yylex()`, nom imposé par **Yacc** pour son analyseur lexical.
- Si nous utilisons **Lex** pour produire l'analyseur lexical, nous remplaçons la routine `yylex()` dans la troisième partie de la spécification **Yacc** par la clause :
  `#include "lex.yy.c`
  et nous spécifions alors chaque action **Lex** de façon qu'elle retourne un terminal connu de **Yacc**.
- En utilisant la clause `#include "lex.yy.c"`, le programme `yylex` a accès aux noms que **Yacc** a donnés aux unités lexicales puisque le fichier de sortie de **Lex** est compilé comme étant une partie du fichier de sortie **Yacc** `y.tab.c`.

Ligne de commande

Sous le système UNIX, si la spécification **Lex** est dans le fichier `lexeur.l` et la spécification **Yacc** dans le fichier `syntaxeur.y`, on compile de cette manière :

```
lex lexeur.l
yacc syntaxeur.y
cc -ly -ll y.tab.c
```

On obtient un exécutable `a.out` qui permet de traduire le langage désiré.

Récupération sur erreur

En **Yacc**, la récupération sur erreur peut s'effectuer en utilisant une forme de productions d'erreur.

- Premièrement, l'utilisateur décide quels sont les non-terminaux « principaux » auxquels sera associée une forme de récupération sur erreur.
- Les choix typiques sont les sous-ensembles de non-terminaux engendrant les expressions, les instructions, les blocs, et les procédures.
- L'utilisateur ajoute alors à sa grammaire des productions d'erreur de la forme $A \rightarrow \mathit{error}\ \alpha$, où A est un non-terminal principal et α une chaîne de symboles grammaticaux éventuellement vide. *error* est un mot clé **Yacc** réservé.
- **Yacc** construira un analyseur à partir d'une telle spécification en traitant les productions d'erreur comme des productions ordinaires.
- Quand l'analyseur produit pat **Yacc** découvre une erreur, il traite de façon spéciale les états dont les ensembles d'items contiennent des productions d'erreur.

Récupération sur erreur (suite)

- Lorsque **Yacc** rencontre une erreur, il dépile les symboles jusqu'à ce qu'il trouve un état en sommet de pile dont dont l'ensemble d'items qui le composent comprend un item de la forme $A \rightarrow \cdot\alpha$.
- L'analyseur décale alors sur la pile une unité lexicale fictive `error`, comme s'il avait rencontré l'unité lexicale `error` dans le flot d'entrée.
- Quand α est ϵ, une réduction vers A est effectuée immédiatement et l'action sémantique associée à la production $A \rightarrow$ *error* (définie par l'utilisateur) est appelée.
- L'analyseur élimine alors les symboles d'entrée jusqu'à en rencontrer un sur lequel l'analyse normale peut reprendre.
- Si α est non vide, **Yacc** élimine un à un les symboles d'entrée à la recherche d'une sous-chaîne qui peut être réduite vers α.
- Si α est uniquement formée de terminaux, il recherche cette chaîn de terminaux dans l'entrée et les réduit en les décalant sur la pile.
- L'analyseur réduira alors *error*α vers A, puis reprendra l'analyse normale.

Exemple

- Si on a la production *instr* $\longrightarrow$ *error*;, alors l'analyseur saute jusqu'après le prochain point-virgule lorsqu'il rencontre une erreur. Il supposera qu'il a reconnu une instruction (modèle *instr*).
- Dans notre code précédent on peut ajouter `lignes : error '\n'`
- L'analyseur suspend son analyse lorsqu'il rencontre une erreur de syntaxe. En découvrant l'erreur, il dépile jusqu'à ce qu'il rencontre un état qui a une action décaler sur l'unité lexicale *error*. L'état 0 en est un exemple car ses items incluent *lignes* $\longrightarrow \cdot$*error* $'\backslash n'$.
- Notons que l'état 0 est toujours en fond de pile.
- L'analyseur décale l'unité lexicale *error* en sommet de pile et commence alors à lire un à un les symboles d'entrée jusqu'à trouver le caractère de fin de ligne.
- À ce moment, l'analyseur décale la fin de ligne sur la pile, réduit *error* $'\backslash n'$ vers *lignes* et emet un diagnostic d'erreur avec `yyerror`.
- La routine **Yacc** spécifique `yyerrok` replace l'analyseur dans le mode normal de fonctionnement.

```
%{
  #include <ctype.h>
%}
%token CHIFFRE

%%

ligne   : expr '\n'         { printf("%d\n",$1); }
        | error '\n'        { yyerror("Erreur..."); yyerrok; } ;
expr    : expr '+' term     { $$ = $1 + $3; } | term ;
term    : term '*' facteur  { $$ = $1 * $3; } | facteur ;
facteur : '(' expr ')'      { $$ = $2;} | CHIFFRE ;

%%

yylex()
{
  int c;
  c = getchar();
  if (isdigit(c))
  {
    yyval = c - '0';
    return CHIFFRE;
  }
  return c;
}
```