Interprétation et compilation

Chapitre 9

Analyse syntaxique

Pablo Rauzy <pr@up8.edu>
pablo.rauzy.name/teaching/ic

Analyse syntaxique

- En compilation, l'*analyse syntaxique* est l'étape qui suit l'*analyse lexicale* et qui précède l'*analyse sémantique*.
- Le rôle de cette analyse est de comprendre et vérifier la *structure* du code source.
- Cette structure est définie par une *grammaire non-contextuelle*.

Grammaires non-contextuelles

- Les grammaires des langages de programmation sont le plus souvent *récursives* :
  - Une expression est
    - un nombre, ou
    - une expression, suivie d'un opérateur, suivi d'une expression.
  - Une instruction est
    - une affectation, ou
    - une accolade ouvrante, suivie d'une suite d'instructions, suivie d'une accolade fermante.
- Les *grammaires non-contextuelles* (de type 2 dans la hiérarchie de Chomsky) sont tout à fait adaptées à ce genre de description.

- Formellement, une grammaire non-contextuelle est un quadruplet $G = (T, N, A, D)$ où
  - T est un ensemble de *symboles terminaux*,
  - N est un ensemble de *symboles non-terminaux*,
  - $A \in N$ est le *symbole de départ* (ou *axiome*),
  - D est un ensemble de *règles de dérivation* (ou *production*) de la forme $S \to S_1 S_2 \cdots S_k$, où $S \in N$ et $S_i \in T \cup N$.
- Dans le cadre de la compilation
  - les symboles terminaux sont les unités lexicales du langage,
  - les symboles non-terminaux sont des *variables syntaxiques* qui désignent des ensembles de chaînes de symboles terminaux,
  - le symbole de départ est un non-terminal qui désigne “un programme”,
  - les règles de dérivations peuvent être vues comme *descriptives* ou *génératives*.

Exemple

- Exemple de grammaire non-contextuelle pour les expressions arithmétiques (avec `+` et `*`) :
  - $T = \{$ nombre, `+`, `*` $\}$,
  - $N = \{E, O, I\}$,
  - $A = E$,
  - $D = \{E \to I, E \to EOE, I \to \text{nombre}, O \to \texttt{+}, O \to \texttt{*}\}$.

Règles de dérivation

- On peut voir les règles de dérivation comme des *règles de réécriture*, dans ce cas elles sont *génératives* au sens où elles engendrent tous les programmes syntaxiquement valides.
  - $S \to S_1 S_2$ se lit “pour produire tous les S possibles il faut d’abord produire tous les S_1 possibles puis tous les S_2 possibles”.
- On peut aussi les voir comme des *règles d’analyse*, auquel cas elles sont *descriptives* au sens où elles permettent de reconnaître les programmes syntaxiquement valides.
  - $S \to S_1 S_2$ se lit “pour reconnaître un S valide il faut d’abord reconnaître un S_1 valide puis un S_2 valide”.

- Vous avez sûrement déjà croisé des grammaires décrites sous forme de BNF.
- C'est une notation très courante pour les grammaires non-contextuelles.
- Attention toutefois :
  - l'ensemble des terminaux et des non-terminaux est implicite dans une description BNF,
  - le symbole de départ est par convention le membre gauche de la première règle.

Exemple et TD

- Voici une grammaire BNF :
  - ```
    <prog>  ::= ( <instr> ";" )*
    <instr> ::= <var> "=" <expr>
    <expr>  ::= "0" | "1"
              | <var>
              | <expr> <bop> <expr>
              | <uop> <expr>
              | "(" <expr> ")"
    <var>   ::= ( "a"-"z" )+
    <bop>   ::= "&&" | "||"
    <uop>   ::= "!"
    ```
    ```

Exemple et TD

- Voici une grammaire BNF :
  - ```
    <prog>  ::= ( <instr> ";" )*
    <instr> ::= <var> "=" <expr>
    <expr>  ::= "0" | "1"
              | <var>
              | <expr> <bop> <expr>
              | <uop> <expr>
              | "(" <expr> ")"
    <var>   ::= ( "a"-"z" )+
    <bop>   ::= "&&" | "||"
    <uop>   ::= "!"
    ```
- Que représente cette grammaire ?
    ```

Exemple et TD

- Voici une grammaire BNF :

```
<prog>  ::= ( <instr> ";" )*
<instr> ::= <var> "=" <expr>
<expr>  ::= "0" | "1"
          | <var>
          | <expr> <bop> <expr>
          | <uop> <expr>
          | "(" <expr> ")"
<var>   ::= ( "a"-"z" )+
<bop>   ::= "&&" | "||"
<uop>   ::= "!"
```

- Que représente cette grammaire ?
- Quels sont ses symboles terminaux ?

Exemple et TD

- Voici une grammaire BNF :

```
<prog>  ::= ( <instr> ";" )*
<instr> ::= <var> "=" <expr>
<expr>  ::= "0" | "1"
          | <var>
          | <expr> <bop> <expr>
          | <uop> <expr>
          | "(" <expr> ")"
<var>   ::= ( "a"-"z" )+
<bop>   ::= "&&" | "||"
<uop>   ::= "!"
```

- Que représente cette grammaire ?
- Quels sont ses symboles terminaux ?
- Quels sont ses symboles non-terminaux ?

Exemple et TD

- Voici une grammaire BNF :

```
<prog>  ::= ( <instr> ";" )*
<instr> ::= <var> "=" <expr>
<expr>  ::= "0" | "1"
          | <var>
          | <expr> <bop> <expr>
          | <uop> <expr>
          | "(" <expr> ")"
<var>   ::= ( "a"-"z" )+
<bop>   ::= "&&" | "||"
<uop>   ::= "!"
```

- Que représente cette grammaire ?
- Quels sont ses symboles terminaux ?
- Quels sont ses symboles non-terminaux ?
- Quel est son symbole de départ ?

- Soit une grammaire non-contextuelle $G = (T, N, A, D)$.
- Soient $S \in N$ et $\sigma \in (T \cup N)*$ tels que $S \to \sigma \in D$.
- $\forall \alpha, \beta \in (T \cup N)*$, $\alpha S \beta$ se dérive en $\alpha\sigma\beta$ en *une étape*, ce qu'on écrit $\alpha S \beta \Rightarrow \alpha\sigma\beta$.
  D'où le nom de grammaire non-contextuelle (ici, α et β sont le contexte de S, et la dérivation de S en σ n'en dépend jamais).
- Si $\sigma_0 \Rightarrow \sigma_1 \Rightarrow \sigma_2 \Rightarrow \cdots \Rightarrow \sigma_n$, on note $\sigma_0 \overset{n}{\Rightarrow} \sigma_n$.
- Si α se dérive en β en un nombre quelconque (y compris zéro) d'étapes, on note $\alpha \overset{*}{\Rightarrow} \beta$.

- Le langage engendré par $G = (T, N, A, D)$ est l'ensemble $L(G) = \left\{ \sigma \in T* \mid A \overset{*}{\Rightarrow} \sigma \right\}$.
- Si $p \in L(G)$ on dit que p est une *phrase* de $L(G)$
  (ou, en compilation, un programme syntaxiquement valide).

Arbres de dérivation

- Toujours avec une grammaire $G = (T, N, A, D)$.
- Soit $\sigma \in T*$, telle que $\sigma \in L(G)$, il existe donc une dérivation $A \overset{*}{\Rightarrow} \sigma$.
- On peut représenter cette dérivation par un arbre, qu'on appelle *arbre de dérivation* :
  - la racine de l'arbre est A,
  - les feuilles sont étiquetées par les symboles terminaux de σ,
  - chaque nœud correspond à une dérivation (donc si $S \to S_1 S_2 \cdots S_n$ est utilisé, le nœud étiqueté par S a n fils étiquetés par S_1, S_2, etc.).

Arbres de dérivation

- Toujours avec une grammaire $G = (T, N, A, D)$.
- Soit $\sigma \in T*$, telle que $\sigma \in L(G)$, il existe donc une dérivation $A \overset{*}{\Rightarrow} \sigma$.
- On peut représenter cette dérivation par un arbre, qu'on appelle *arbre de dérivation* :
  - la racine de l'arbre est A,
  - les feuilles sont étiquetées par les symboles terminaux de σ,
  - chaque nœud correspond à une dérivation (donc si $S \to S_1 S_2 \cdots S_n$ est utilisé, le nœud étiqueté par S a n fils étiquetés par S_1, S_2, etc.).

→ Dessinez l'arbre de dérivation de "`foo = 1 && !bar; o = !(foo || 0);`" pour la grammaire donnée par la BNF vue précédemment.

- L'arbre de dérivation qui est nécessairement construit par l'analyseur syntaxique pour vérifier la validité de la syntaxe du programme est une information précieuse.
- En effet, elle correspond à la *structure* du code source.
- C'est donc une première étape significative dans la compréhension du code.

- Considérons la grammaire suivante :
  - ```
    <expr> ::= <expr> "||" <expr>
             | <expr> "&&" <expr>
             | "(" <expr> ")"
             | <val>
    <val>  ::= "0" | "1"
    ```
- Ainsi que la phrase suivante : “`1 && 0 || 1`”.
    ```

Ambiguïté

- Considérons la grammaire suivante :
  - ```
    <expr> ::= <expr> "||" <expr>
             | <expr> "&&" <expr>
             | "(" <expr> ")"
             | <val>
    <val>  ::= "0" | "1"
    ```
- Ainsi que la phrase suivante : “`1 && 0 || 1`”.
- Il y a plusieurs dérivations possibles, dont :
  - $\texttt{E} \Rightarrow \texttt{E \&\& E} \Rightarrow \texttt{E \&\& E || E} \overset{*}{\Rightarrow} \texttt{V \&\& V || V}$
  - $\texttt{E} \Rightarrow \texttt{E || E} \Rightarrow \texttt{E \&\& E || E} \overset{*}{\Rightarrow} \texttt{V \&\& V || V}$

→ Dessinons les arbres de dérivations correspondants.
    ```

Ambiguïté

- Considérons la grammaire suivante :
  - ```
    <expr> ::= <expr> "||" <expr>
             | <expr> "&&" <expr>
             | "(" <expr> ")"
             | <val>
    <val>  ::= "0" | "1"
    ```
- Ainsi que la phrase suivante : “`1 && 0 || 1`”.
- Il y a plusieurs dérivations possibles, dont :
  - $\texttt{E} \Rightarrow \texttt{E \&\& E} \Rightarrow \texttt{E \&\& E || E} \overset{*}{\Rightarrow} \texttt{V \&\& V || V}$
  - $\texttt{E} \Rightarrow \texttt{E || E} \Rightarrow \texttt{E \&\& E || E} \overset{*}{\Rightarrow} \texttt{V \&\& V || V}$

→ Dessinons les arbres de dérivations correspondants.

- Cette grammaire est donc *ambigüe*.
    ```

Équivalence

- Soient deux grammaires G_1 et G_2.
- Si $L(G_1) = L(G_2)$ on dit que les deux grammaires sont *équivalentes*.
- Il est souvent possible de remplacer une grammaire ambigüe par une grammaire non-ambigüe équivalente.
  - Il n'y a pas de méthode générale pour cela, donc ne n'est pas automatisable.

Exemple

- Considérons la grammaire suivante :
  - ```
    <expr> ::= <expr> "||" <conj>
             | <conj>
    <conj> ::= <conj> "&&" <term>
             | <term>
    <term> ::= "(" <expr> ")" | <val>
    <val>  ::= "0" | "1"
    ```
- Elle est équivalente à la précédente, mais est non-ambigüe.
- La phrase “`1 && 0 || 1`” ne peut se dériver que de la façon suivante :
  - `E` $\Rightarrow$ `E || C` $\Rightarrow$ `C || C` $\Rightarrow$ `(C && T) || C` $\overset{*}{\Rightarrow}$ `V && V || V`
- Arrivez-vous à vous convaincre que c’est la seule façon de dériver cette phrase ?

→ Dessinons l’arbre de dérivation correspondant.

→ Que remarquez-vous ?
    ```

Factorisation à gauche

- La *factorisation à gauche* est une technique pour corriger certains cas d'ambiguïté.
- Prenons les règles de dérivation $S \to \sigma\alpha$ et $S \to \sigma\beta$.
- Il est possible de les réécrire en $S \to \sigma S'$, $S' \to \alpha$ et $S' \to \beta$.
- Exemple :
  - ```
    <expr> ::= "if" <expr> "then" <expr>
             | "if" <expr> "then" <expr> "else" <expr>
    ```

  devient :

  - ```
    <expr>       ::= "if" <expr> "then" <expr> <maybe-else>
    <maybe-else> ::= "else" <expr> | ε
    ```

- Si on décide que la dérivation vide ne doit se faire qu'en dernier recours si aucune autre n'est possible (dérivation gloutonne), alors il n'y a plus d'ambiguïté.

→ Dérivez la phrase "`if foo then if bar then baz else quux`" de toutes les façons possibles dans les deux grammaires.

Grammaires contextuelles

- Un exemple simple de langage plus puissant que ce que peut décrire une grammaire non-contextuelle est $Lc = \{ww \mid w \in (a|b)*\}$.

Grammaires contextuelles

- Un exemple simple de langage plus puissant que ce que peut décrire une grammaire non-contextuelle est $Lc = \{ww \mid w \in (a|b)*\}$.
- En effet, le seul moyen d'obtenir ce langage serait avec une grammaire de type $G = (\{a, b\}, \{S, A, B, A', B'\}, S, D)$ où D contient les règles de dérivation suivantes :
  - $S \to aAS$
  - $S \to bBS$
  - $S \to aA'$
  - $S \to bB'$
  - $Aa \to aA$
  - $Bb \to bB$
  - $Ab \to bA$
  - $Ba \to aB$
  - $AA' \to A'a$
  - $BB' \to B'b$
  - $AB' \to A'b$
  - $BA' \to B'a$
  - $aA' \to aa$
  - $bB' \to bb$
  - $aB' \to ab$
  - $bA' \to ba$
- Cette grammaire est *contextuelle* (cf les membres gauches des règles de dérivation).

Grammaires contextuelles

- Un exemple simple de langage plus puissant que ce que peut décrire une grammaire non-contextuelle est $Lc = \{ww \mid w \in (a|b)*\}$.
- En effet, le seul moyen d'obtenir ce langage serait avec une grammaire de type $G = (\{a, b\}, \{S, A, B, A', B'\}, S, D)$ où D contient les règles de dérivation suivantes :

• $S \to aAS$	• $Aa \to aA$	• $AA' \to A'a$	• $aA' \to aa$
• $S \to bBS$	• $Bb \to bB$	• $BB' \to B'b$	• $bB' \to bb$
• $S \to aA'$	• $Ab \to bA$	• $AB' \to A'b$	• $aB' \to ab$
• $S \to bB'$	• $Ba \to aB$	• $BA' \to B'a$	• $bA' \to ba$

- Cette grammaire est *contextuelle* (cf les membres gauches des règles de dérivation).
- On ne pourra pas décrire de tels langages lors de l'analyse syntaxique.

→ Pensez-vous que c'est gênant pour la compilation ? Pourquoi ?

Analyse descendante

- Le principe d'un *analyseur descendant* est de construire l'arbre de dérivation en partant de la racine (le symbole de départ de la grammaire) et en allant vers les feuilles (les symboles terminaux).

- On suppose maintenant qu'on dispose d'une grammaire non-contextuelle non-ambigüe et factorisée à gauche.

- On lit les unités lexicales de gauche à droite (L pour *left-to-right*).
- On cherche systématiquement à faire une *dérivation gauche* (L pour *leftmost*).
- On ne regarde qu'un seul lexème d'avance pour décider quelle règle appliquer (1).

- On implémente ce type d'analyseur avec une pile.
- On commence par l'initialisation :
  - on empile le symbole de départ de la grammaire,
  - l'unité lexicale courante est la première.
- Ensuite on fait en boucle :
  - si le sommet de la pile est un terminal σ :
    - si l'unité lexicale courante correspond à σ : on dépile et on avance d'un cran (l'unité lexicale suivante devient la courante),
    - sinon signaler une erreur "σ attendu" ;
  - si le sommet de la pile est un non-terminal S :
    - si il n'y a qu'une seule règle de dérivation $S \to S_1 S_2 \cdots S_k$: dépiler S et empiler S_k, S_{k-1}, ..., S_1,
    - sinon choisir la seule règle qui convient en fonction de l'unité lexicale courante et appliquer le même traitement.
- L'algorithme termine lorsque la pile est vide :
  - si l'unité lexicale courante est `EOF`, on a réussi à reconnaître une phrase du langage (en compilation, un programme syntaxiquement valide),
  - sinon signaler une erreur "unité lexicale truc inattendue".

Exemple

- Déroulons ensemble l'analyse descendante de "`1 && 0 || 1`" dans notre grammaire désambiguée et factorisée à gauche :
  - ```
    <expr>       ::= <disj> <disj_suite>
    <disj>       ::= <conj> <conj_suite>
    <disj_suite> ::= "||" <expr>
                   | ε
    <conj>       ::= "(" <expr> ")"
                   | <val>
    <conj_suite> ::= "&&" <expr>
                   | ε
    <val>  ::= "0" | "1"
    ```
    ```

- On peut implémenter ce type d'analyseur sur le modèle de l'algorithme énoncé précédemment.
- C'est à dire de la même manière que les automates qu'on a vu pour les analyseurs lexicaux : avec une fonction de transition générique et une table de transitions dépendante de la grammaire.
- La différence est que cette fois on ajoute une pile.

- On peut aussi se servir de la pile d'appels de notre langage de programmation au lieu de gérer la pile nous-même.
- Dans ce cas, le code de l'analyseur sera complètement lié à la grammaire.
- Pour chaque règle de dérivation $S \to w$ (où $w \in (T \cup N)*$) écrire une fonction `match-S` qui consiste :
  - si il n'y a qu'une seule règle avec ce membre gauche, à appeler successivement les fonctions `match-*` correspondantes à w,
  - sinon à faire un switch sur l'unité lexicale courante pour faire la même chose sur la bonne règle.
- Pour chaque terminal σ une fonction `match-σ` qui :
  - si l'unité lexicale courante est σ passe à l'unité lexicale suivante,
  - sinon signale l'erreur "σ attendu".

→ Regardons ensemble une telle implémentation (`ll1.ml`).

Analyse ascendante

- Le principe d'un *analyseur ascendant* est de construire l'arbre de dérivation en partant des feuilles (les symboles terminaux de la phrase à analyser) et en allant vers la racine (les symboles de départ de la grammaire).

- L'idée est à nouveau d'utiliser une pile, mais "dans l'autre sens".
- Cette fois-ci, on empile les symboles terminaux lus, et c'est quand les symboles du haut de la pile forment le membre droit d'une règle de dérivation qu'on les dépile et empile le non-terminal membre gauche de cette règle.
- Quand on a fini de lire la phrase à analyser, l'analyse a réussi si la pile ne contient plus que le symbole de départ.
- L'empilement s'appelle un *décalage* (ou *shift*).
- Le dépilement s'appelle une *réduction* (ou *reduce*).

- L'idée est à nouveau d'utiliser une pile, mais "dans l'autre sens".
- Cette fois-ci, on empile les symboles terminaux lus, et c'est quand les symboles du haut de la pile forment le membre droit d'une règle de dérivation qu'on les dépile et empile le non-terminal membre gauche de cette règle.
- Quand on a fini de lire la phrase à analyser, l'analyse a réussi si la pile ne contient plus que le symbole de départ.
- L'empilement s'appelle un *décalage* (ou *shift*).
- Le dépilement s'appelle une *réduction* (ou *reduce*).

→ Voyez-vous un problème possible avec cette idée ?

- Il y a deux problèmes de déterminisme avec cette méthode :
  - Si le haut de la pile correspond aux membres droits de plusieurs règles de dérivation, laquelle choisir ?
  - Si le haut de la pile correspond au membre droits d'au moins une règle de dérivation, doit-on continuer à décaler pour en trouver un autre ? ou réduire tout de suite ?

- Si la grammaire est factorisée à gauche, le premier problème ne devrait pas se poser.
- Le second problème se pose chaque fois qu'une réduction est possible : doit-on l'appliquer ou faire un décalage ?
  - La solution est de faire en sorte que l'*analyse par décalage-réduction* corresponde à la construction en sens inverse d'une dérivation à droite.

→ Reprenons ensemble notre exemple avec "`1 && 0 || 1`".

- Ce qu'on est en train de construire est un *analyseur LR(k)*.

- On lit les unités lexicales de gauche à droite (L pour *left-to-right*).
- On cherche systématiquement à faire une *dérivation droite* (R pour *rightmost*).
- On regarde au maximum k lexèmes à la fois (k).
  - Quand on ne le précise pas, c'est que k vaut 1.

Construction

- En fait, ces analyseurs peuvent être beaucoup plus efficace que les analyseurs descendants.
- En revanche, ils nécessitent une table de transitions signalant, en fonction de la pile et de l'unité lexicale courante, quelle est la bonne action à faire entre décaler et réduire (en précisant dans ce cas suivant quelle règle de dérivation).
- La construction de cette table à la main serait extrêmement longue et fastidieuse, et particulièrement prône à l'erreur tout en étant difficile à débugguer...

Générateurs d'analyseurs syntaxiques – `yacc`

- L'outil `yacc` a été originalement écrit au début des années 1970 par Stephen C. Johnson en langage B (mais a très vite été réécrit en C).
- La version libre du projet GNU s'appelle Bison.
- Comme `lex`, un outil équivalent existe pour de nombreux langages.

- En fait, l'outil **yacc** n'est pas strictement capable de générer des analyseurs syntaxiques LR(1).
- Il est un peu moins puissant : il génère des analyseurs syntaxiques LALR(1), c'est à dire LA(1)LR(0) (LA signifie "look ahead").
- La différence est subtile : l'analyseur peut regarder la prochaine unité lexicale fournie par l'analyseur lexical, mais il ne les utilise pas au moment de construire les états de l'automate.

- `https://ocaml.org/manual/lexyacc.html`

→ Regardons ensemble un exemple d'analyseur syntaxique (`parser.mly`).

Associativité et priorité des opérateurs

- Les analyseurs LALR étant moins puissants que les LR, il arrive que ceux-ci soient coincés par des conflits shift/reduce ou des conflits reduce/reduce.
- Pour la plupart des langages de programmation, ces conflits peuvent être résolus en explicitant la priorité et le type d'associativité de certains terminaux.
  - Cela peut être fait implicitement par l'ordre d'apparition des règles ou explicitement dans la description de la grammaire donnée à l'outil **yacc**.