

Compilateurs : Analyse syntaxique

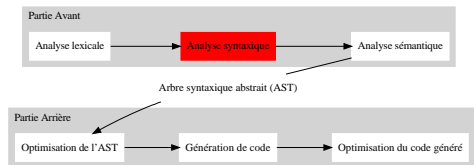
Matthieu Amiguet

2008 – 2009



“Vous êtes ici”

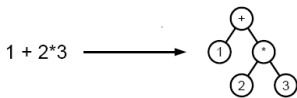
2



Analyse syntaxique

3

- À la sortie de l'analyse lexicale, on a séparé et identifié des lexèmes...
- ... mais ceux-ci sont toujours complètement "à plat"
- On aimerait maintenant pouvoir comprendre la *structure* du programme
- En d'autres termes, on a séparé les *mots*, mais on aimerait connaître la *structure grammaticale* de la phrase



- Ce traitement s'appelle "Analyse syntaxique" (en : *Parsing*)

Comment faire ?

4

- Les méthodes d'analyse syntaxique s'apparentent à celles d'analyse lexicale :
 - On se donne un moyen de décrire formellement le langage à analyser
 - Éventuellement, on génère automatiquement un analyseur à partir de cette description
- Mais
 - Ici, on ne se contente plus d'une réponse oui/non, on veut récupérer la structure !
 - Les expressions régulières ne sont plus suffisantes !
- On donc donc devoir commencer par étudier un nouveau type de description de langages.

1 Grammaires (E)BNF et arbres syntaxiques

- Grammaires et arbres d'analyse
- Techniques d'analyse

2 Implémenter un analyseur syntaxique

Analyse syntaxique
Grammaires (E)BNF et arbres syntaxiques
Grammaires et arbres d'analyse

Limites des expressions régulières 6

- Les expressions régulières sont bien pratiques, mais elles ne permettent pas de décrire n'importe quel langage
- Notamment, elles ne permettent pas d'exprimer les structures imbriquées
 - Vérifier le nombre de parenthèses ouvrantes et fermantes dans $((12+(5*2)/7)-(4*3))+1$
- Les structures imbriquées étant centrales dans de nombreux langages de programmation, on va donc devoir trouver un nouveau formalisme.

Analyse syntaxique
Grammaires (E)BNF et arbres syntaxiques
Grammaires et arbres d'analyse

Exemple 7

- Lexèmes : Det(le), Det(la), Nom(chat), Nom(souris), Verbe(mange)
- Symboles intermédiaires : S, GN, GV
- Symbole de départ : S
- Règles
 - $S \rightarrow GN\ GV$
 - $GN \rightarrow Det\ Nom$
 - $GV \rightarrow Verbe\ GN$
- On "voit" que cette grammaire permet de décrire des phrases comme "Le chat mange la souris", "La souris mange le chat"...
 - ... mais aussi "La chat mange le souris", ...

Analyse syntaxique
Grammaires (E)BNF et arbres syntaxiques
Grammaires et arbres d'analyse

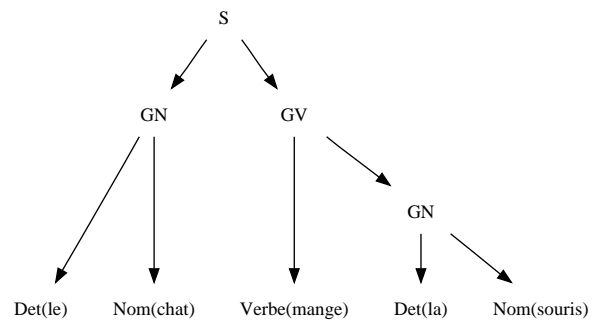
Grammaires (E)BNF 8

- Une grammaire en Forme de Backus-Naur (BNF) est composée de
 - Une liste de lexèmes, aussi appelés *terminaux*
 - Une liste de symboles intermédiaires, aussi appelés *non-terminaux*, dont un *symbole de départ*
 - Un ensemble de règles de la forme
 - Non-terminal \rightarrow suite de terminaux et non-terminaux
- Les alternatives peuvent être notées avec "|"
 - $GV \rightarrow Verbe\ GN\ | Verbe$
- On peut aussi accepter les *, ? et + avec la même signification que pour les expressions régulières ; on parle alors de Forme de Backus-Naur Étendue, ou EBNF
- On est donc très proche d'une description régulière, mais *la récursivité est autorisée !*

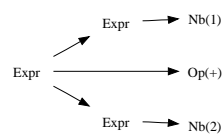
- On appelle *arbre d'analyse* ou *arbre syntaxique* un arbre dont les noeuds sont étiquetés par des symboles grammaticaux de telle manière que
 - les feuilles sont étiquetées par des terminaux et les noeuds intérieurs par des non-terminaux
 - la racine est étiquetée par le symbole de départ de la grammaire
 - les fils d'un noeud interne étiqueté par N correspondent aux membres d'un des choix de N, dans l'ordre
 - les terminaux étiquetant les feuilles correspondent à la suite de lexèmes d'entrée, dans l'ordre.

- Dans certains cas, une grammaire peut admettre plusieurs arbres d'analyse différents pour une même chaîne d'entrée
- Exemple :
 - $expression \rightarrow nombre \mid - expression \mid (expression) \mid expression \text{ op } expression$
 - Chaîne d'entrée : $1+2*3$
- Idéalement, les grammaires ne devraient jamais être ambiguës...
- ... mais ceci est parfois difficile à obtenir
- On peut donc se donner des moyens de lever l'ambiguïté avec des informations complémentaires.

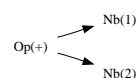
Chaîne d'entrée : "le chat mange la souris"



Parfois, les arbres syntaxiques contiennent bien plus d'information que nécessaire :



alors qu'une forme simplifiée suffirait :



- Un arbre tiré de l'arbre d'analyse mais dont la structure est simplifiée pour ne contenir que les éléments pertinents est appelé *arbre abstrait* (ou *arbre syntaxique abstrait*, *AST*)
- Que doit contenir un arbre abstrait ?
 - Il n'y a pas de règle...
 - ... un arbre abstrait doit contenir, assez précisément, ce qui est utile pour la suite !
 - C'est donc un choix de conception.

- Il existe essentiellement deux grandes approches pour l'analyse syntaxique
- On peut partir du symbole de départ et essayer de construire l'arbre en direction des feuilles : *Analyse descendante*
 - Un noeud est toujours construit *avant* tous ses enfants
- On peut partir des feuilles et les organiser progressivement en arbre : *Analyse ascendante*
 - Un noeud est toujours construit *après* tous ses enfants.

- Dans les deux types d'analyse, une solution simple serait d'essayer toutes les combinaisons possibles jusqu'à trouver une analyse correcte du programme...
- ... Mais la complexité algorithmique serait bien trop grande !
 - On vise si possible une complexité linéaire...
- On se permet généralement de regarder un ou deux lexèmes à l'avance et de prendre des décisions définitives sur cette base
 - Moins puissant, donc nécessite d'écrire des grammaires qui s'y prêtent...
 - ... Mais *beaucoup* plus efficace !

- 1 Grammaires (E)BNF et arbres syntaxiques
- 2 Implémenter un analyseur syntaxique
 - Descente récursive et analyse LL(1)
 - Analyse ascendante LALR(1)

Écriture ou génération ?

17

- Comme pour les analyseur lexicaux, il existe des générateurs de code
 - qui prennent en entrée une description de la grammaire ainsi que quelques informations annexes, et
 - qui produisent en sortie le code d'un analyseur syntaxique
- Avantages similaires au cas de l'analyse lexicale
 - Souplesse en cas de changement de la grammaire
 - Moins d'erreurs
 - Algorithmes optimisés.

Nombreuses méthodes

19

- Il existe de très nombreuses méthodes pour réaliser l'analyse syntaxique – ascendante comme descendante
- Chacune représente un compromis entre
 - La puissance (nombre de grammaires pour lesquelles "ça marche")
 - La rapidité
 - La mémoire nécessaire
- Nous allons voir
 - Comment écrire un analyseur descendant "LL(1)" à la main
 - Comment fonctionne un analyseur ascendant (généralisé) "LALR(1)"

Le plus connu : Yacc/Bison

18

- Générateur d'analyseur le plus largement utilisé
- Génère des analyseurs ascendants (plus précisément, LALR(1))
- Yacc : *Yet another compiler compiler*
 - Mais ce n'est pas vraiment un compilateur de compilateur !
- Bison : version GNU, produit du C ANSI
- Comme pour lex, ce programme a été imité, modifié, etc. de nombreuses fois pour différents langages.

Descente récursive

20

- Étant donné une grammaire adéquate, il est assez facile d'écrire à la main un analyseur récursif descendant
- Prenons par exemple


```
input → expression EOF
expression → term rest_expression
term → IDENTIFIÉRIER | parenth_expr
parenth_expr → '(' expr ')'
rest_expr → '+' expression | ε
```
- On peut voir cette grammaire d'une manière opérationnelle :
 - "je peux reconnaître une expression en commençant par reconnaître un terme, mais alors je dois nécessairement avoir un rest_expression après"...

```

def input():
    return expression() and require(token('EOF') ←
    )

def expression():
    return term() and require(rest_expression())

def term():
    return token('identifiant') or parenth_expr()

def parenth_expr():
    return token('(') and require(expression()) ←
    and require(token(')'))

def rest_expression():
    return (token('+') and require(expression()) ←
    ) or True

```

La fonction `token` permet de regarder si le prochain lexème est du type requis et le cas échéant de le “consommer”

```

def token(t):
    global LATok # Look-Ahead Token
    if LATok.type != t:
        return False
    LATok = lex.token()
    return True

```

La fonction `require` permet de signaler une erreur si quelque chose ne s'est pas passé comme prévu

```

def require(found):
    if found:
        return True
    error("Error while parsing near token %s!" % ←
    LATok.value)

```

```

** parsing: 12+3
12  input:
12  | expression:
12  | | term:
12  | | | token: identifiant
+   | | rest_expression:
+   | | | token: +
3   | | | expression:
3   | | | | term:
3   | | | | | token: identifiant
EOF | | | | rest_expression:
EOF | | | | | token: +
EOF | | | | | ** Failed!
EOF | token: EOF
** result: True

```

```

** parsing: 12++3
12  input:
12  |  expression:
12  |  |  term:
12  |  |  |  token: identifieur
+   |  |  |  rest_expression:
+   |  |  |  |  token: +
+   |  |  |  |  expression:
+   |  |  |  |  |  term:
+   |  |  |  |  |  |  token: identifieur
+   |  |  |  |  |  |  ** Failed!
+   |  |  |  |  |  |  parenth_expr:
+   |  |  |  |  |  |  |  token: (
+   |  |  |  |  |  |  |  ** Failed!
+   |  |  |  |  |  |  |  ** Failed!
+   |  |  |  |  |  |  |  ** Failed!
+   |  |  |  |  |  |  |  ** Failed!
Error while parsing near token +!

```

- Cette méthode permet donc de reconnaître la syntaxe en ne prenant les décisions que sur la base d'un seul lexème en avance...
- ... pour autant que la grammaire s'y prête
 - Contre-exemple : grammaire récursive à gauche
 $E \rightarrow E '+' \text{identifieur}$
- Cet algorithme, en version légèrement optimisée, s'appelle LL(1)
- Une grammaire qui "marche" avec cet algorithme est dite LL(1)
 - Problème : peu de grammaires sont LL(1) !

- On part des feuilles et on construit l'arbre en remontant vers la racine
- Il existe plusieurs variantes de l'algorithme : LR(0), LR(1), SLR(1), LALR(1), ...
- La méthode la plus utilisée est LALR(1)
 - bonne combinaison de puissance, d'efficacité et d'utilisation mémoire
- LALR(1) est souvent utilisé par les générateurs de code à la Yacc/Bison

- On consomme les lexèmes de la chaîne d'entrée, de gauche à droite, et on les pousse sur une pile
- Deux types d'action
 - avancer (*shift*) : on retire le premier lexème de l'entrée et on l'empile
 - réduire (*reduce*) : si le sommet de la pile contient la partie droite d'une règle, on dépile les lexèmes correspondants et on empile la partie gauche de la règle.
- Ce type d'algorithme est aussi appelé *shift-reduce parsing*.

Exemple

29

- Grammaire :
 - $E \rightarrow E+E \mid E * E \mid id$
- Analyse de $x+y*z$

Conflits

31

- Si une réduction est possible, mais que le *look-ahead token* ne permet pas d'exclure une autre réduction, on a un *conflit avancer-réduire (shift-reduce conflict)*
- Si deux réductions différentes sont possibles, on a un *conflit réduire-réduire (reduce-reduce conflict)*
- Exemple : à l'étape 6 de l'exemple ci-dessus, on a un *conflit avancer-réduire*
- Les conflits sont un signe d'une ambiguïté dans la grammaire
 -

Détails

30

- Quand une réduction n'est pas possible et qu'on peut encore avancer, on **AVANCE**
- Lorsque les éléments au sommet de la pile correspondent à la partie droite d'une règle, ET qu'il n'existe aucune règle qui utiliserait le sommet de la pile *plus le look-ahead token* (premier lexème de l'entrée), on **RÉDUIT**

Résolution de conflits

32

- Modifier la grammaire pour la rendre non-ambigüe
 - On aura donc aussi un arbre syntaxique modifié
 - Solution la plus "propre", mais pas toujours la plus facile. . .
- Se donner des lignes de conduites, par exemple
 - Préférer avancer à réduire
 - Entre deux réductions possibles, prendre la première donnée par le programmeur
- Indiquer des priorités sur les lexèmes
 - "La multiplication est plus prioritaire que l'addition"
 - Marche bien pour les expressions

- Dans l'algorithme présenté ci-dessus, on cherche à chaque étape si une règle correspond au sommet de la pile
- Ça marche, mais c'est bien évidemment beaucoup trop lourd !
- Dans les faits un algorithme LALR(1) pré-calculé toutes les possibilités et les stocke dans des tables
 - Notre algorithme devient alors une sorte de grosse machine d'état avec une chaîne d'entrée et une pile. . .
- La construction de ces tables sort du cadre de ce cours, mais les générateurs font ça très bien pour nous !

En plus des références générales sur les compilateurs :

- D. Grune et C.J.H. Jacobs , "Parsing Techniques – A Practical Guide", Ellis Horwood, 1990
- Première édition disponible en ligne à l'adresse <http://www.cs.vu.nl/~dick/PTAPG.html>
 - 320 pages sur les techniques d'analyse syntaxique expliquées en détail et très progressivement. . .
 - La seconde édition, publiée en 2008 chez Springer, comporte même 664 pages !
