Compilation : du source à l'éxécutable

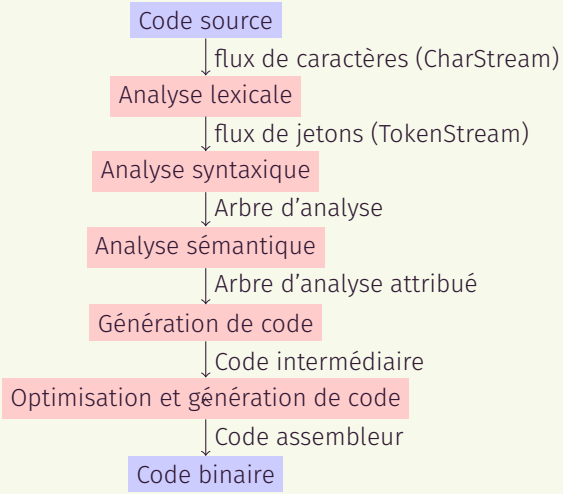
[L3 Informatique] Théorie des langages et compilation

Gaétan Richard

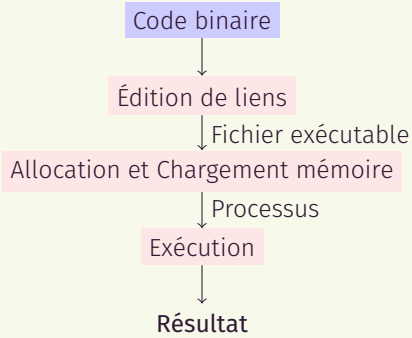
2024-2025

Vue d’ensemble

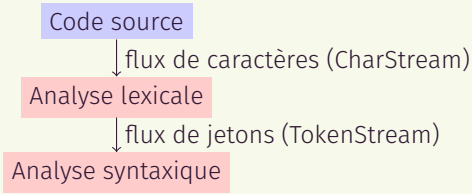
Les grandes étapes de la compilation ...



Jusqu'à l'exécution



Analyse lexicale


Code source
flux de caractères (CharStream)
Analyse lexicale
flux de jetons (TokenStream)
Analyse syntaxique

Analyse lexicale : vocabulaire et définition

Lexème : chaîne de caractères correspondant à une unité élémentaire du texte.

Unité lexicale : classe ou type de lexèmes. *Exemples* : mot-clé, identifiant, nombre, opérateur arithmétique, ...

Jeton (token) : objet ayant :

- une unité lexicale ;
- un lexème ;
- un numéro de ligne (et de caractère) dans le source ;
- une valeur pour un nombre, adresse dans la table des symboles pour un identificateur ;
- ...

L'analyse lexicale converti un fichier d'entrée en un flux de jetons.

Fonctionnement de l'analyse lexicale

La définition des jetons se fait en général par des **expressions régulières**.

L'analyseur lexical est un **automate fini** avec des actions qui permet de découper les jetons.

Gestion des ambiguïtés

- 1 lexème satisfait 2 expressions
- 1 lexème est le préfixe d'un autre

Gestion des ambiguïtés

1 lexème pour 2 expressions

Par exemple, si un mot est à la fois identifié comme un `mot clé` et comme un `identifiant`.

Gestion via des priorités (en général, ordre d'apparition dans le code) du lexeur.

1 lexème est le préfixe d'1 autre

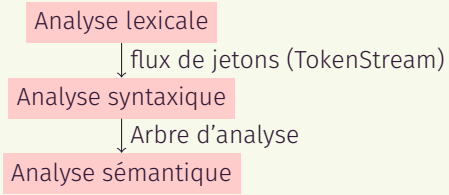
Par exemple, si un préfixe d'un identifiant est un mot clef.

Par défaut une **approche gloutonne** (*greedy*), on conserve la plus longue correspondance. Il existe aussi souvent une notation pour prendre la plus petite correspondance (`*?`). Ceci est très utile pour les commentaires et les chaînes de caractères.

Exemple Antlr

```
lexer grammar Calc0 ;
// Lexer
NUMBER: ('0'..'9')+ ;
ST: 'S';
ID: ('a'..'z'|'A'..'Z')('a'..'z'|'0'..'9')*;
PLUS: '+';
MOINS: '-';
COMMENT : '//' ~('\n') -> skip ;
UNMATCH: . -> skip;
```

Analyse syntaxique


Analyse lexicale
flux de jetons (TokenStream)
Analyse syntaxique
Arbre d'analyse
Analyse sémantique

Analyse syntaxique : vocabulaire et définition

Une grammaire donne la syntaxe des mots admissibles.

L'arbre d'analyse a pour nœuds des non-terminaux et pour feuilles des terminaux (ce sont les jetons du lexeur).

L'arbre de syntaxe abstraite (AST) a pour nœuds des opérations et pour feuilles les opérandes.

Principe : l'analyse syntaxique transforme un flux de jetons en arbre d'analyse ou en AST.

Fonctionnement de l'analyse syntaxique

Cas général : il est impossible de faire un analyseur syntaxique qui transforme efficacement un flux de jetons en arbre d'analyse pour n'importe quelle grammaire.

Mais pour des grammaires particulières, on peut garantir d'être efficace. `Antlr` offre cette garantie pour les grammaires LL* .

grosso modo, on ne fait pas de backtrack lorsqu'on cherche une règle : il suffit de regarder à distance k vers l'avant pour être fixé sur la règle qu'on doit applique

Travail dans l'analyse syntaxique

- gérer les ambiguïtés : s'il existe deux arbres de dérivations correspondant à une expression, on utilise des informations de **priorité** ou d'**associativité**.
- Gérer l'incorrect : si la grammaire n'est pas dans la bonne forme, il existe un certain nombre de mécanismes.

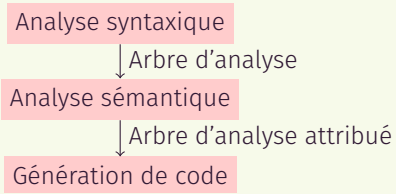
Exemple Antlr

```
grammar Calc1 ;

// Parser ...
calcul
  : expr calcul
  | ST ID expr calcul
  ;
expr
  : NUMBER
  | ID
  | PLUS expr expr
  | MOINS expr expr
  ;
```

```
// Lexer
NUMBER: ('0'..'9')+;
ST: 'S';
ID: ('a'..'z'|'A'..'Z')('a'..'z'|'0'..'9')*;
PLUS: '+';
MOINS: '-';
```

Analyse sémantique


Analyse syntaxique
Arbre d'analyse
Analyse sémantique
Arbre d'analyse attribué
Génération de code

Fonctionnement

Objectif : Donner un sens à l'arbre de dérivation.

Méthode : ajouts d'annotations dans l'arbre calculées avec des règles locales.

Informations :

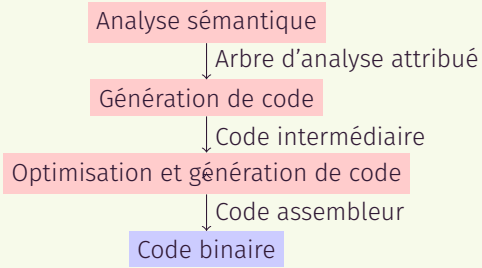
- vérification des types ;
- résolution des noms (via une table des symboles) ;
- affectation ;
- ...

Exemple Antlr

```
grammar Calc2 ;
@members { int store = 0; }
// Parser
calcul
  : expr calcul { System.out.println($expr.val); }
  | ST ID expr calcul { store = $expr.val;}
  | EOF
  ;
expr returns [int val]
  : NUMBER { $val = $NUMBER.int ;}
  | ID { $val = store;}
  | PLUS a=expr b=expr { $val = $a.val + $b.val ;}
  | MOINS a=expr b=expr { $val = $a.val - $b.val;}
  ;

//Lexer ...
```

Génération du code


Analyse sémantique
Arbre d'analyse attribué
Génération de code
Code intermédiaire
Optimisation et génération de code
Code assembleur
Code binaire

Objectif : passer de l'AST à du code machine.

Représentation intermédiaire : on utilise une représentation intermédiaire avant le code machine binaire

Avantages :

- Indépendance de la machine physique ;
- Phase d'optimisation plus facile.

Exemple : code 3 adresses.

Si on suppose que l'AST reflète la priorité et l'associativité des opérateurs :

- il suffit de parcourir l'arbre ;
- affecter un nouveau registre pour chaque résultat d'opération ;
- à chaque nœud, on récupère le code machine et le registre contenant le résultat ;
- on obtient le code complet par un parcours postfixe (Gauche - Droite - Racine).