Notes de cours

GIF 340 : ÉLÉMENTS DE COMPILATION

Chapitre 1

Introduction aux langages et à la compilation

Ahmed KHOUMSI

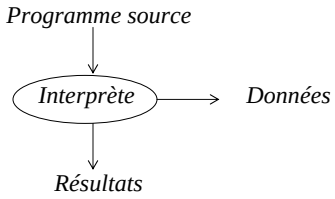
Deux approches pour traiter un programme source

Première approche : Interprétation

Un interpréteur accède au programme source pour :

- Lire
- Analyser
- Exécuter

} une instruction après l 'autre



Exemples : LISP, PROLOG

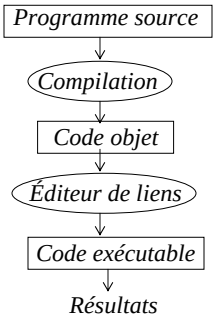
On peut aussi considérer MATLAB comme un interpréteur, où les les instructions sont des commandes.

Autre exemple : Shell de UNIX

Deuxième approche : Compilation

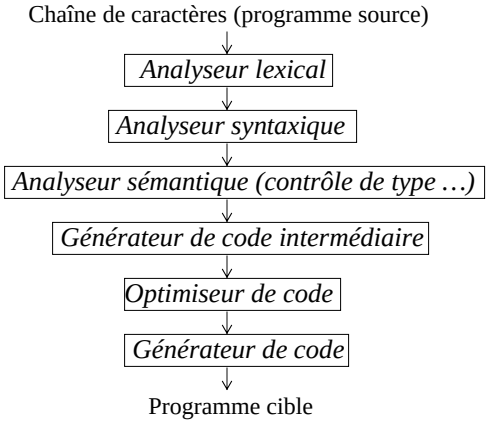
Le programme source est entièrement traduit en instructions compréhensibles (et donc exécutables) par un ordinateur

Étapes avant exécution du programme



Exemples : PASCAL, C, C++, FORTRAN, ADA, SMALLTALK

Phases d 'un compilateur



Analyseur lexical (scanner)

Flots de caractères constituant programme source :
- lus de gauche à droite
- groupés en unités lexicales
- caractères superflus supprimés

Unité lexicale (token) = suite de caractères ayant signification collective

But essentiel de l'analyse lexicale est :
- générer des unités lexicales
- déterminer si chaque unité lexicale est un mot du vocabulaire du langage utilisé

Exemple : Instruction en Pascal *vitesse* := *vitesse_initiale* +*accélération**40
Le scanner regroupe les caractères dans les unités suivantes :

Identificateur	*vitesse*	Identificateur	*accélération*
Symbole d 'affectation	:=	Symbole de multiplication	*
Identificateur	*vitesse_initiale*	Nombre	40
Symbole d 'addition	+		

Analyseur lexical (suite)

Si l 'analyseur rencontre un mot non autorisé alors il génère une erreur

Exemple : Si $ n 'est pas utilisé dans le langage considéré alors
un message d'erreur est généré lorsque $ est rencontré

Remarque : Le scanner ne travaille pas sur plusieurs unités lexicales à la fois

Exemple : Langage Pascal
Soit la chaîne de caractères := +**toto*-45
Cette chaîne est jugée correcte par le scanner qui reconnaît les unités lexicales suivantes, qui sont correctes lorsqu'on les considère séparément :
:= + * *toto* - 45

Analogie : Langue française
Soit la phrase *tu manger des fruits*
Cette phrase n 'est évidemment pas correcte grammaticalement, mais le vocabulaire qu 'elle contient est correct.
Une analyse lexicale n 'y détecte donc aucune erreur.

Analyseur syntaxique (parser)

But du Parser est de :
- regrouper les unités lexicales en structures grammaticales correctes
- déterminer si la syntaxe (ou GRAMMAIRE) est correcte

Grammaire d'un langage définie par ensemble de règles (possiblement récursives)

Exemple : Soit l'instruction *vitesse* := *vitesse_initiale* + *accélération**40

Les règles qu 'on peut utiliser sont les suivantes :

Pour la définition des expressions :

Règle 1 : Tout *identificateur* est une expression

Règle 2 : Tout *nombre* est une expression

Règle 3 : Si *expression1* et *expression2* sont des expressions, alors :

expression1 + *expression2*
expression1 * *expression2*
(*expression1*)
} sont des expressions

Pour définition d 'une instruction

Règle 4 : *identificateur* := *expression* est une instruction

Analyseur syntaxique (suite)

Règle 1 implique :

vitesse
vitesse_initiale
accélération
} sont des expressions

Règle 2 implique :

40 est une expression

Règle 3 implique :

accélération * 40
vitesse_initiale + *accélération* * 40
} sont des expressions

(on suppose ici qu 'il y précédence de * sur +)

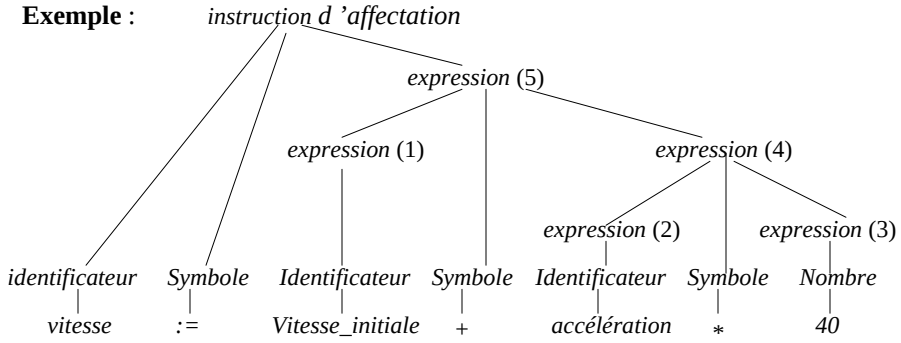
Règle 4 implique :

vitesse := *vitesse_initiale* + *accélération**40 est une instruction

Analyseur syntaxique (suite)

L'analyse syntaxique peut être représentée par un arbre syntaxique de dérivation

Exemple :



L'analyseur lexical construit les feuilles de l 'arbre, qui sont des unités lexicales
L'analyseur syntaxique construit :

- les expressions 1 et 2 à l 'aide de la règle 1
- l 'expression 3 à l 'aide de la règle 2
- les expressions 4 et 5 à l 'aide de la règle 3

Analyseur syntaxique (suite)

Si l 'analyseur syntaxique rencontre une structure qui n 'est pas définie par les règles de la grammaire, alors il signale une erreur.

Exemple : *vitesse* := *vitesse_initiale* +) *accélération**40

- La règle 4 définit une instruction d 'affectation telle que le membre à droite de := est une expression
- Les règles 1, 2 et 3 ne permettent pas de construire la partie à droite de :=

Le parser détecte alors une erreur

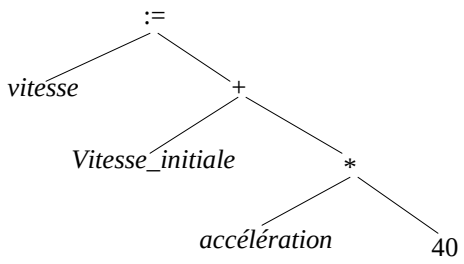
Cela se traduit par le fait qu 'on ne peut pas construire d 'arbre syntaxique de dérivation

Analyseur syntaxique (suite)

Il existe une représentation des structures des phrases qui est plus concise que les arbres syntaxiques de dérivation. Il s 'agit des **arbres syntaxiques abstraits** (en anglais : Abstract Syntax Tree, AST).

Dans un AST :
- les opérateurs sont nœuds
- les opérandes sont des feuilles

Exemple : *vitesse* := *vitesse_initiale* + *accélération**40



Analyseur sémantique (contextuel)

Le but essentiel est de déterminer si le programme contient des erreurs sémantiques statiques. L 'analyseur sémantique vérifie, entre autres :

Les types des identificateurs :

Il détecte par exemple des erreurs du genre :
- opération sur deux types incompatibles
  exemple : ensemble + fichier
  fonction * tableau
- nombre réel utilisé comme indice d 'un tableau. Exemple : tab[2.3]

Il peut aussi effectuer certaines « corrections » sur des types de variables.
exemple : entier + réel
l'entier est converti en réel avant d 'effectuer l 'addition

La portée des identificateurs :

Une erreur est détectée lorsque, par exemple, une variable est :
- d 'une part, déclarée à l 'intérieur d 'une procédure et
- d 'autre part, utilisée à l 'extérieur de la procédure

Analyseur sémantique (suite)

Unicité des identificateurs :

Exemples :

- Une variable doit être déclarée une seule fois (dans une même portée)
- Les étiquettes dans une instruction case (switch) doivent être distincts
- Les éléments d 'un type *énumération* ne peuvent pas être répétés

Flot d 'exécution :

Exemple : Une instruction *break* termine l 'exécution de l 'instruction *while*, *for* ou *switch* qui l 'englobe au plus près
Il y a erreur si une telle instruction (*while*, *for* ou *switch*) n 'existe pas alors qu 'une instruction *break* est utilisée

Parmi les différents contrôles effectués, le contrôle de type est le plus complexe. Il sera étudié en détail ultérieurement.

Analyseur sémantique (suite)

Exemple de contrôle de type :

vitesse := *vitesse_initiale* + *accélération**40

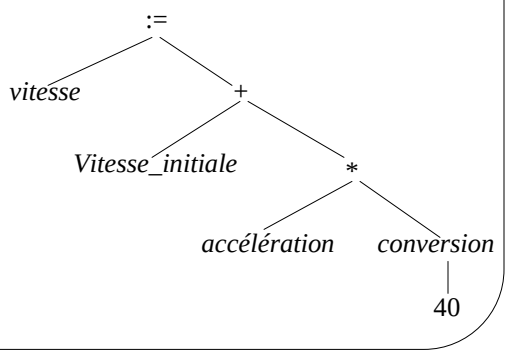
Supposons que *vitesse*, *vitesse_initiale* et *accélération* sont des réels

L 'analyseur sémantique (contrôleur de type) :

- détecte multiplication entre réel (accélération) et nombre entier (40)
- convertit nombre 40 en un réel 40.0

Le parser génère un arbre syntaxique abstrait (AST) qui est ensuite complété par l 'analyseur sémantique.

Pour notre exemple, le parser génère l 'AST de la page 11, et le contrôleur de type (type checker) y insère une conversion pour obtenir l 'AST suivant :



Génération de code intermédiaire

Après l 'analyse sémantique, certains compilateurs produisent une représentation intermédiaire du code source. Celle-ci doit :

- être indépendante de la machine cible
  (contrairement aux langages assembleurs)
- décrire en détail les séquences d 'opérations à effectuer
  (comparable à l 'assembleur)

C 'est au fait un code exécutable d 'une machine abstraite. Ce code doit être :

- d 'un côté, facile à produire, et
- d 'un autre côté, facile à traduire en langage cible.

Génération de code intermédiaire (suite)

Exemples de représentations intermédiaires :

Exemple 1 : AST (nous l 'étudierons en détail ultérieurement)

Exemple 2 : Code à trois adresses (en anglais : Three-Adress Code, TAC)
Dans chaque instruction, il y a :

- au plus un opérateur (binaire ou unaire)
- au plus trois opérandes
- une affectation

Variables temporaires utilisées pour stocker les valeurs calculées

Par exemple, pour l 'instruction *vitesse* := *vitesse_initiale* + *accélération**40 nous obtenons l 'AST de la page 14 et le TAC suivant :

t1 :=*conversion*(40)
t2 := *accélération**t1
t3 :=*vitesse_initiale* + t2
vitesse := t3

Optimisation de code

Amélioration du code intermédiaire selon les critères suivants :
- le temps : durée d ’exécution à minimiser
- la mémoire : occupation de la mémoire à minimiser

Les deux critères étant incompatibles, il faut faire des compromis

Exemple : *vitesse* := *vitesse_initiale* + *accélération**40
- La conversion de 40 en un réel peut être effectuée une fois pour toutes, lors de la compilation.
  Les instructions { t1 :=*conversion*(40) ; t2 :=*accélération**t1 } deviennent t1 :=*accélération**40.0
- Considérons les deux instructions { t3 :=*vitesse_initiale* + t2 ; *vitesse* :=t3 }
  Si ultérieurement aucune opération n ’est effectuée sur t3, alors les 2 instructions peuvent être remplacées par : *vitesse* := *vitesse_initiale* + t2

Les 4 instructions TAC de la page précédente deviennent alors :

t1 :=*accélération**40.0
vitesse := *vitesse_initiale* + t1

Production de code

Code cible en langage assembleur ou machine

Le code engendré peut :
- être exécutable, ou
- nécessiter une édition de liens avec des librairies

Cette phase dépend directement de la machine sur laquelle tournera le programme
En effet, ce code :
- est constitué d ’instructions exécutables par le µp de la machine cible
- utilise des registres dont dispose la machine cible

Exemples :

- Intel 80386, 80486, …
- Motorola 68020, 68030 ...

Parties frontale et finale

Phases d ’un compilateur peuvent être regroupées en deux parties

Partie frontale : contient les cinq phases suivantes :
- analyse lexicale
- analyse syntaxique
- analyse sémantique
- génération de code intermédiaire
- optimisation de code

Cette partie: - dépend : { du code source / du code intermédiaire

- ne dépend pas de la machine cible

Pour sa conception, on ne se soucie pas des détails de la machine sur laquelle le programme est sensé tourner

Parties frontale et finale (suite)

Partie finale : génération de code cible :

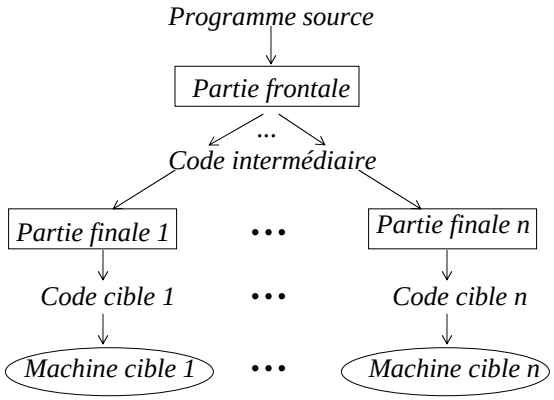
Cette partie: - ne dépend pas du code source

- dépend { du code intermédiaire / de la machine cible

Pour sa conception, on ne se soucie pas du langage source utilisé

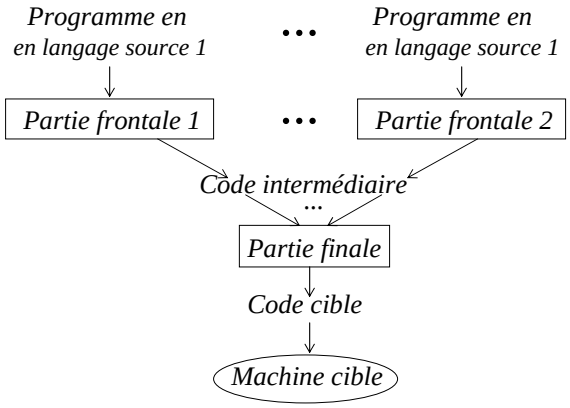
Avantages à effectuer séparation en deux parties

Premier avantage : lorsqu ’on change de machine, seule la partie finale doit être modifiée

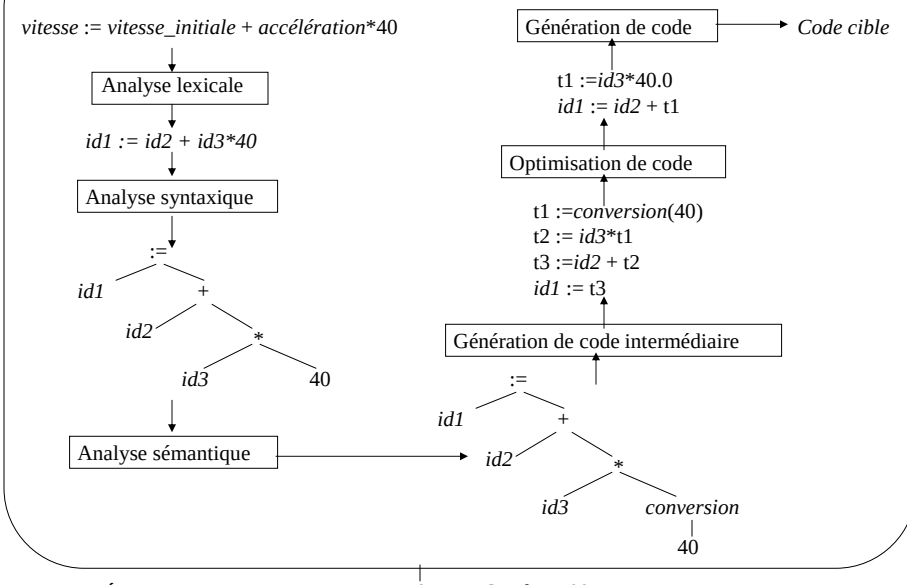


Avantages à effectuer séparation en deux parties (suite)

Second avantage : lorsqu ’on change de langage source, seule la partie frontale doit être modifiée

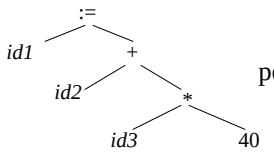


Récapitulation des différentes phases



Implantation de la structure de données engendrée par l 'analyseur syntaxique

L 'AST



peut être implanté comme suit :