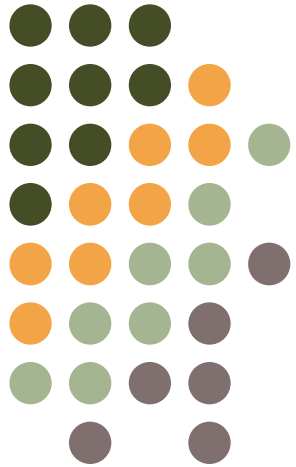
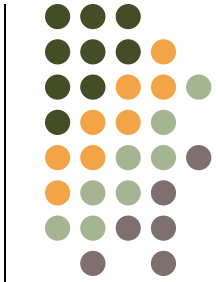
Introduction à la compilation

Dr. Aida Lahouij

Aida.lahouij@gmail.com





Plan

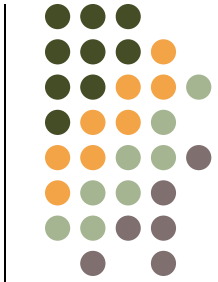
Introduction

- Langage interprété vs langage compilé
- Interpréteur
- Compilateur
- Compilateur hybride
- Préprocesseur
- Assembleur

La compilation

- Analyse lexicale
- Analyse syntaxique
- Analyse sémantique
- Génération de code intermédiaire
- Optimisation de code
- Génération de code
- Gestion de la table des symboles
- Gestion des erreurs

Introduction



- Tout programme écrit dans un « langage de haut niveau » (Pascal, C, C++, Ada…) doit être traduit en instructions exécutables par l'ordinateur

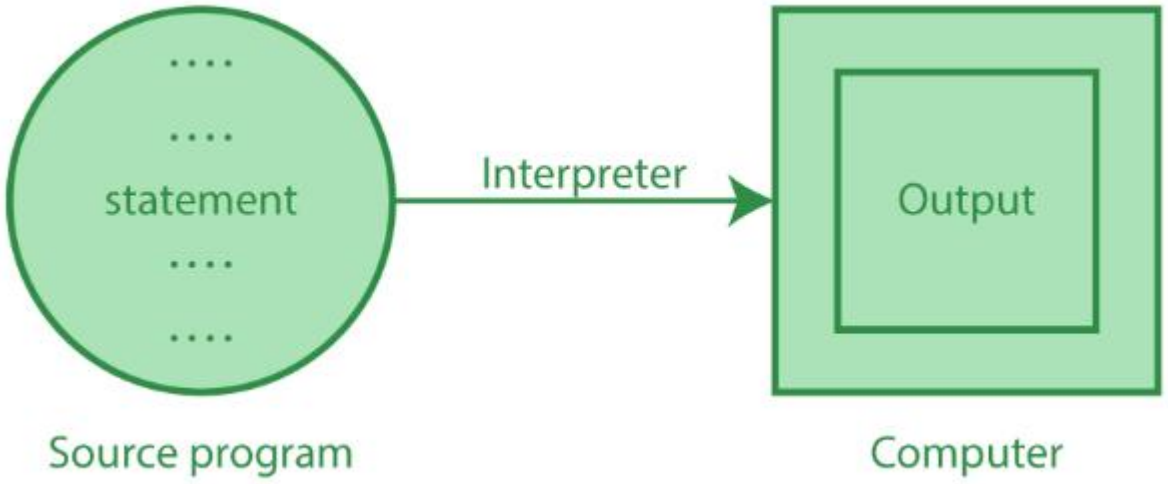
- Cette traduction peut être effectuée soit par un **compilateur** soit par un **interpréteur**

Introduction

Langage interprété vs langage compilé

Les langages interprétés

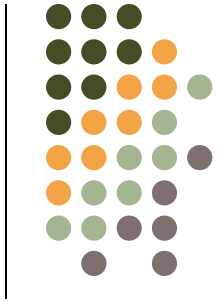
- **Les langages interprétés**, le programmeur peut exécuter immédiatement son programme, voir sans délais les résultats et les éventuelles erreurs.
- **Les langages interprétés**, vous permettent d'exécuter le programme à tout moment, alors que vous l'écrivez.



Introduction

Langage interprété vs langage compilé

Les langages compilés



- **Les langages compilés** s'exécutent plus **rapidement** et sont mieux appropriés au développement d'applications commerciales, mais requièrent beaucoup **plus d'efforts et d'apprentissage.**
- **Les langages compilés** nécessitent des étapes supplémentaires, la compilation et la liaison, avant que le programmeur ne puisse exécuter son œuvre.

Les compilateurs traduisent l'ensemble du programme soit

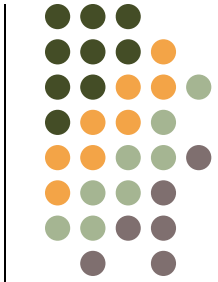
- en langage d'une machine **concrète**

ou

- en langage d'une machine **abstraite JVM,**

Introduction

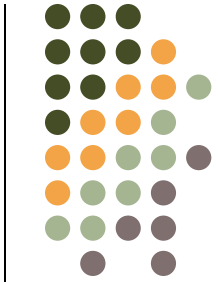
Les interpréteurs



Fonctionnement :

- Traite le code ligne par ligne.
- Lit, analyse, et prépare chaque instruction pour le processeur dans l'ordre d'apparition.
- Utilise ses bibliothèques internes pour convertir le code source en commandes machine.
- Transmet directement les instructions au processeur après conversion.

Introduction



Les interpréteurs

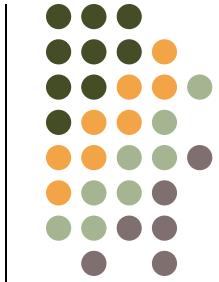
! Note

Parmi les langages de programmation les plus célèbres ayant majoritairement recours à un interpréteur pour la conversion du code source en code machine, on compte notamment

BASIC, **Perl**, **Python**, **Ruby** et **PHP**.

Ces langages sont d'ailleurs souvent réunis sous le terme de « **langages interprétés** ».

Introduction



Les compilateurs

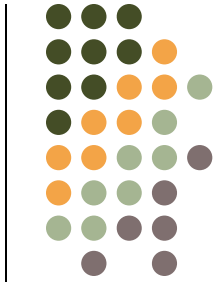
Définition :

- **Programme** qui traduit tout le **code source** en **code machine** avant son exécution.

Fonctionnement :

- Effectue une **traduction complète et préalable** du code source en instructions lisibles par le processeur.
- Le processeur exécute ensuite le programme à l'aide du code machine généré.
- Assure que toutes les instructions nécessaires sont disponibles pour traiter les données et produire les résultats.

Introduction



Les compilateurs

Étape Intermédiaire dans la Compilation

Code intermédiaire :

- Avant la traduction en code machine, le code source est souvent converti en code intermédiaire (ou code objet).
- Ce code est **adapté à différentes plateformes** et peut être utilisé par un interpréteur.

Organisation des instructions :

- Les compilateurs déterminent l'ordre d'exécution des instructions pour optimiser leur transmission au processeur.

!Note

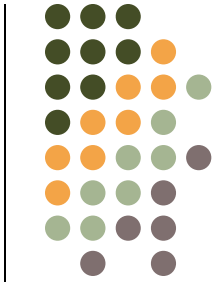
Parmi les langages entièrement compilés, on compte notamment des piliers tels que **C**, **C++** et **Pascal**

Introduction

Interpréteurs vs compilateurs

Aspect	Interpréteur	Compilateur
Exécution	Immédiate, sans étape préalable de compilation.	Nécessite une traduction complète avant exécution.
Lancement	Plus rapide.	Plus lent (compilation nécessaire avant exécution).
Développement	Simplifié grâce au débogage **ligne par ligne**.	Débogage possible uniquement après compilation.
Performance	Plus lent pendant l'exécution (utilise la puissance de calcul).	Plus rapide une fois lancé (aucun besoin du compilateur).
Ressources nécessaires	Nécessite l'interpréteur pendant toute l'exécution.	Le compilateur n'est plus requis après compilation.

Introduction



Solution hybride: le compilateur à la volée

Solution Hybride : Compilateur Just-In-Time (JIT)

Principe :

- Combine les avantages du compilateur et de l'interpréteur.
- Traduction du code pendant l'exécution, à la manière d'un interpréteur.

Avantages :

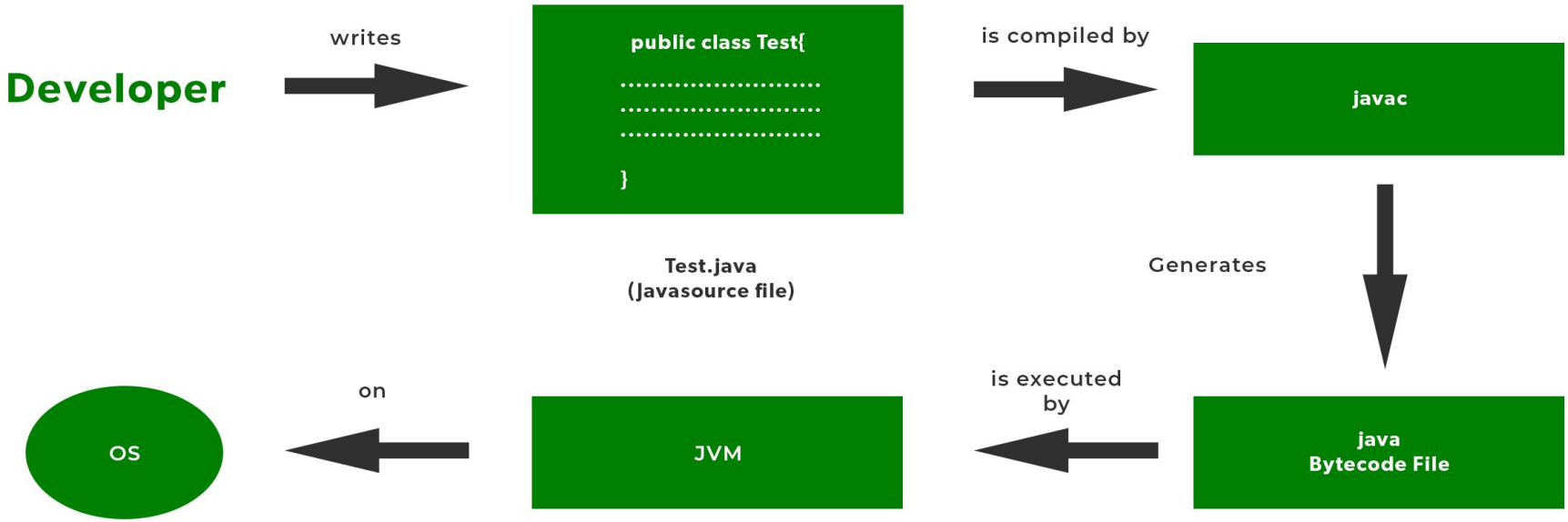
- Vitesse d'exécution élevée (comme un compilateur).
- Développement simplifié (comme un interpréteur).

Exemple :

- Java utilise la compilation à la volée pour améliorer les performances.
- Le code objet est converti en code machine au moment de l'exécution.

Introduction

Solution hybride: le compilateur à la volée

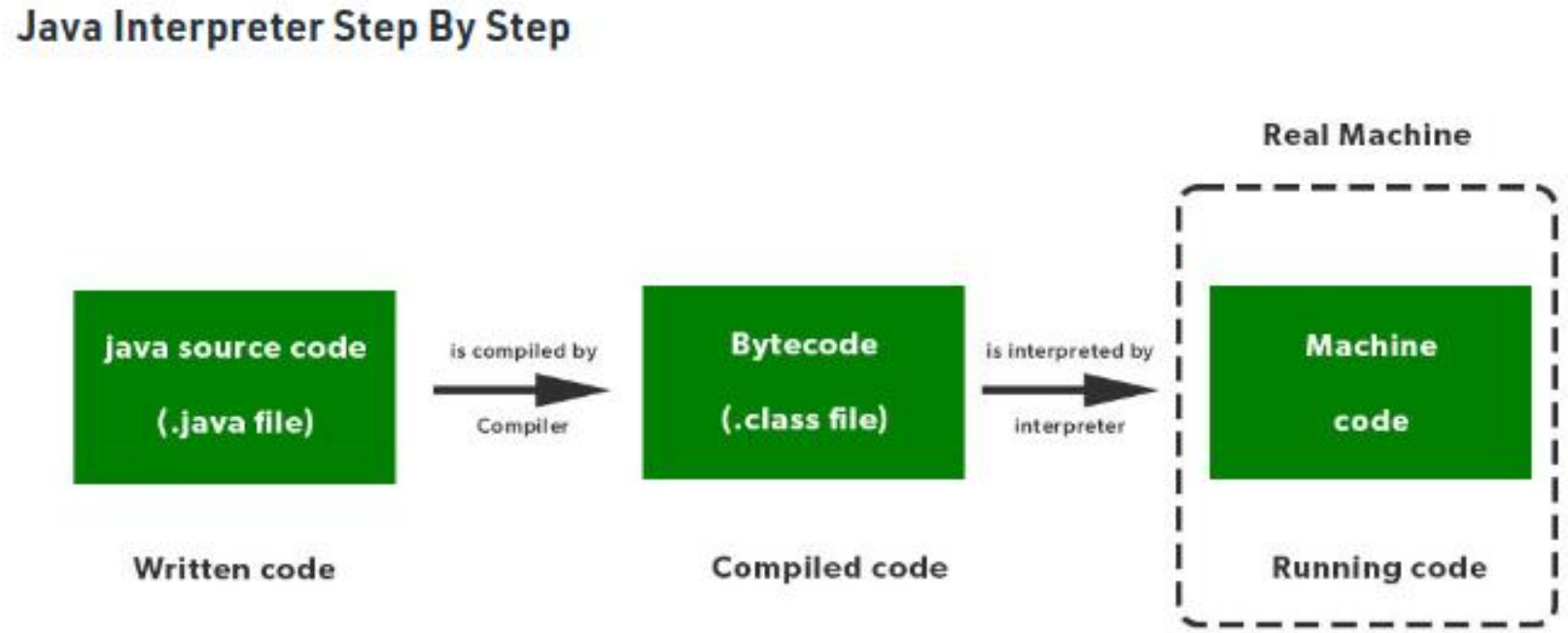


Step from java.code to machine code

- java source code is compiled.
- Transformed to bytecode by java compiler.
- Interpreted and executed by the java virtual machine on the underlying operating system.

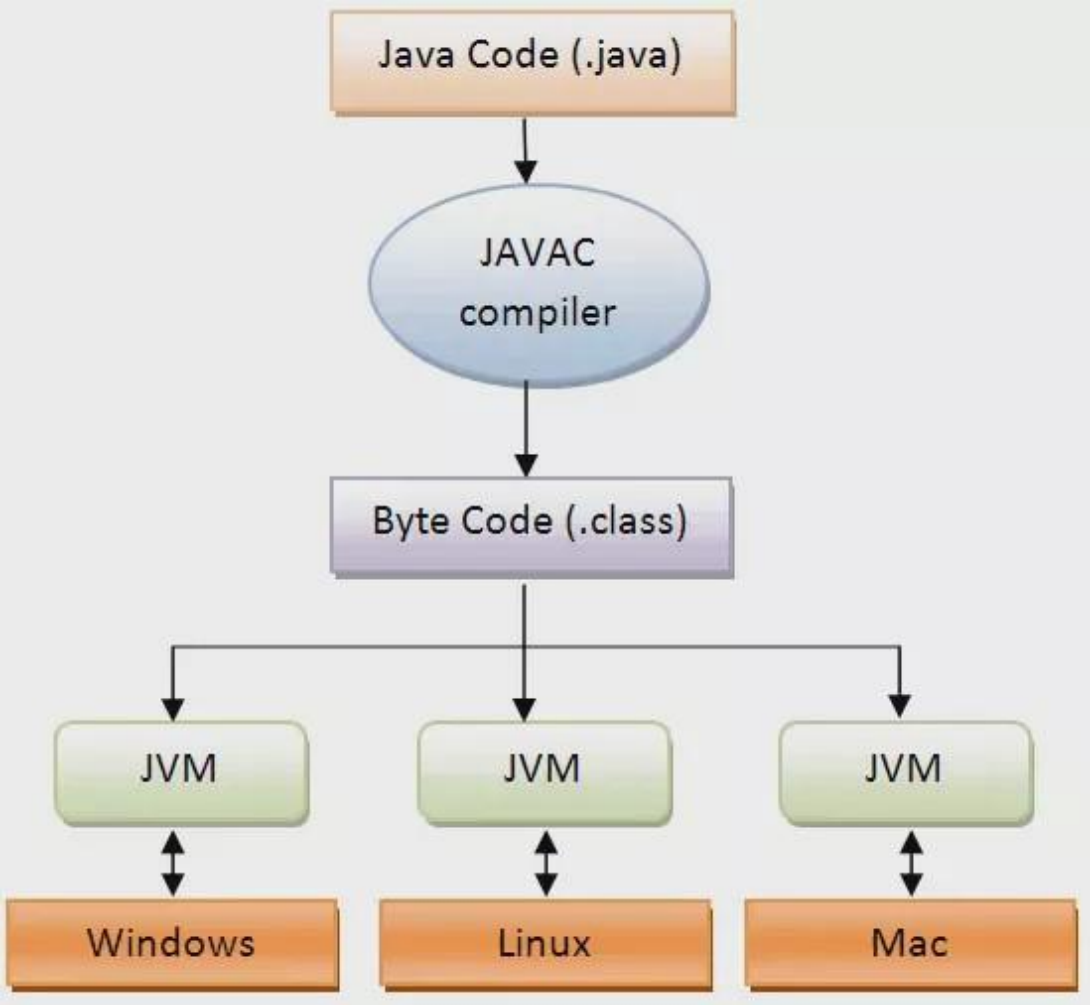
Introduction

Solution hybride: le compilateur à la volée



Introduction

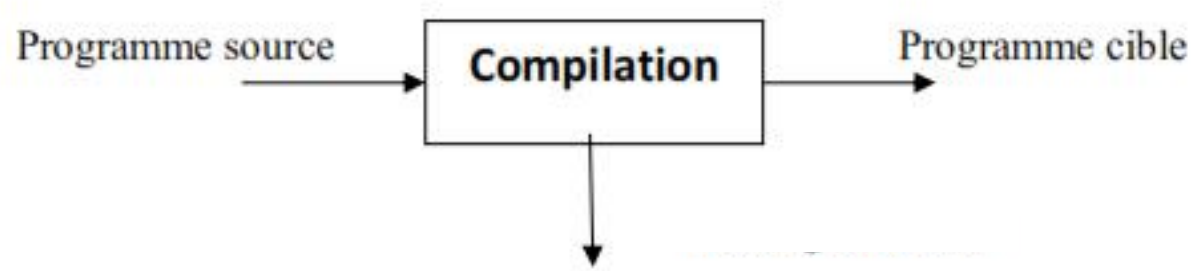
Solution hybride: le compilateur à la volée

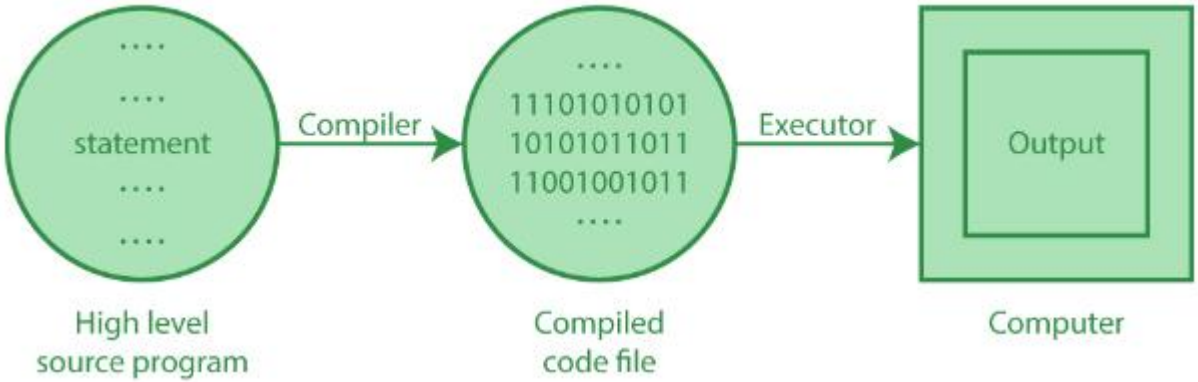


Introduction

La compilation

- **La compilation** transpose le programme du langage dans lequel il a été écrit au langage natif de l'ordinateur.





Introduction

Définition (Compilateur)

Un compilateur est un programme informatique de traduction $C^{L_C}_{L_S \to L_O}$ avec

1. *L_C le langage avec lequel est écrit le compilateur*
2. *L_S le langage source à compiler*
3. *L_O le langage cible ou langage objet*

Exemple (pour $C^{L_C}_{L_S \to L_O}$)

L_C	L_S	L_O
C	Assembleur RISC	Assembleur RISC
C	C	Assembleur P7
C	java	C
Java	$LaTeX$	HTML
C	XML	PDF

Si $L_C = L_S$: peut nécessiter un bootstrapping pour compiler $C^{L_C}_{L_C \to L_O}$

- Le **bootstrapping** décrit en informatique les techniques fondées sur l'écriture d'un compilateur (ou d'un assembleur) dans le langage de programmation cible

Introduction

Bootstrapping : Définition et Application

Définition :

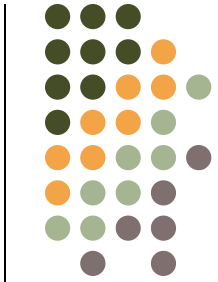
- Le bootstrapping est un processus dans lequel **un système se développe ou s'améliore** de manière autonome, en utilisant des outils ou des techniques plus simples au début, pour aboutir à un système plus complexe.

Dans le contexte des compilateurs :

Créer un compilateur à l'aide d'un compilateur existant.

Introduction

Bootstrapping : Définition et Application



Dans le contexte des compilateurs :

Créer un compilateur à l'aide d'un compilateur existant.

Exemple : Utilisation d'un compilateur simple pour en créer un plus performant, qui à son tour est utilisé pour créer d'autres compilateurs.

Étapes du processus :

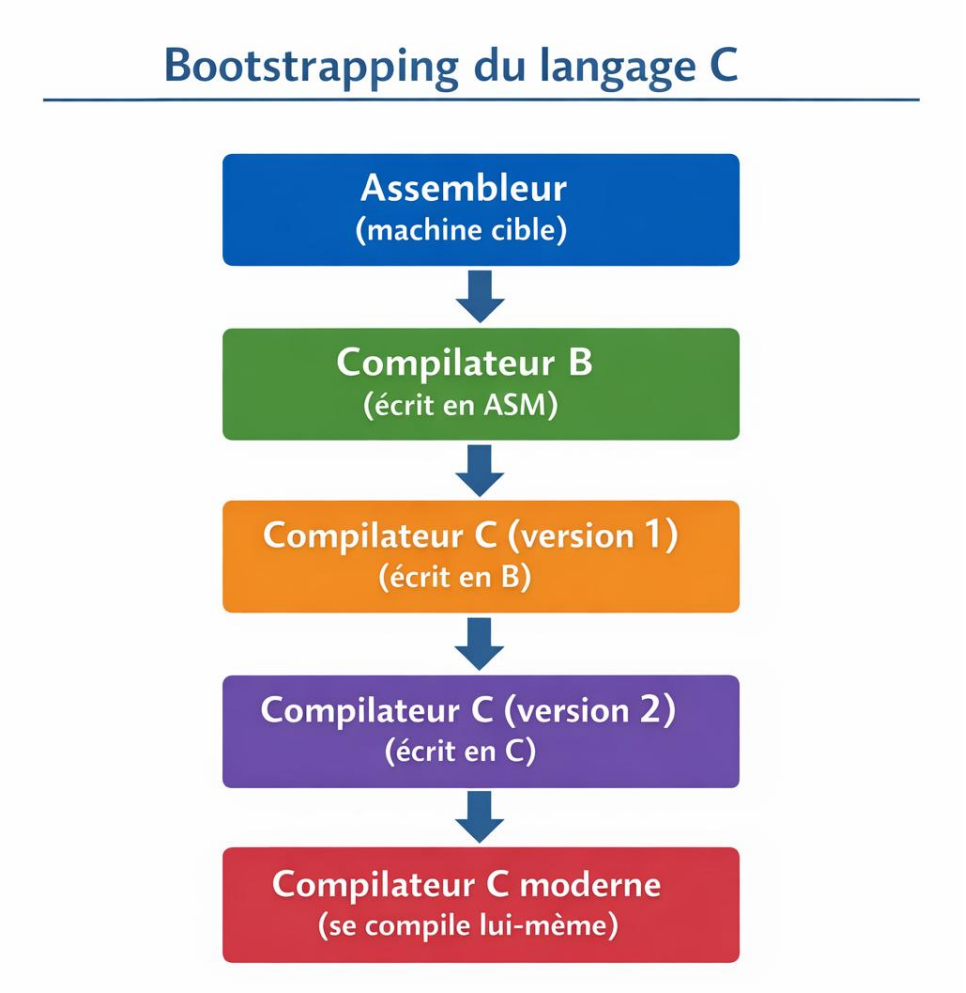
- Initialisation avec un compilateur basique.
- Amélioration progressive du système.
- À terme, le système devient capable de créer des versions plus avancées de lui-même.

Avantage :

Permet de créer des systèmes complexes de manière évolutive et indépendante.

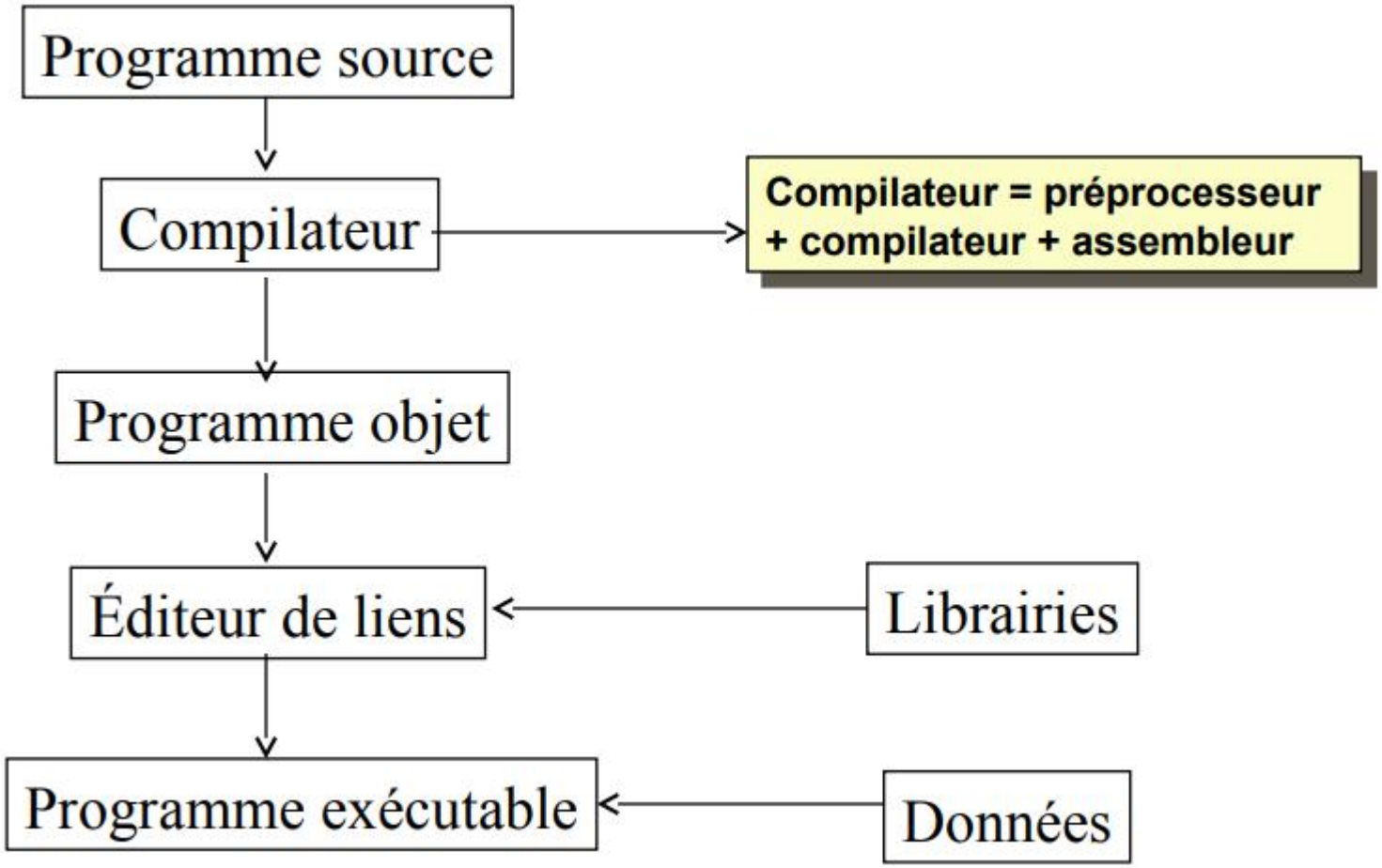
Introduction

Bootstrapping : Définition et Application



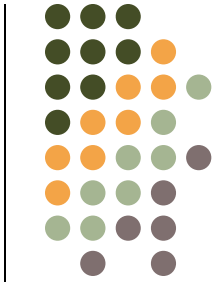
Introduction

La compilation



Introduction

Préprocesseur



- Le préprocessing (ou préprocesseur) est la première étape du processus de compilation.

- Il s'agit d'un traitement automatique du code source avant même que le compilateur ne commence à générer le code machine.

- Son rôle principal est d'appliquer des **transformations spécifiques,** comme **l'insertion** de code, la **suppression** de parties inutiles, ou la **substitution de macros**, pour rendre le code source prêt pour l'étape de traduction (compilation ou interprétation).

Introduction

Préprocesseur

Que fait le préprocesseur ?

- Les fonctions d'un préprocesseur varient selon le langage, mais elles incluent généralement :

- En C, le préprocesseur s'occupe principalement de directives qui commencent par #, comme :

1. Inclusion de fichiers :

→ Remplace la directive par le contenu du fichier inclus..

```
#include <stdio.h>
#include "monheader.h"
```

Introduction

Préprocesseur

Fonctions principales d'un préprocesseur

- Les fonctions d'un préprocesseur varient selon le langage, mais elles incluent généralement :

2. Substitution de macros :

- Remplace des symboles par des valeurs ou des blocs de code définis par l'utilisateur.

Exemple :

```
#define PI 3.14159
```

- Dans le code source, toutes les occurrences de PI seront remplacées par 3.14159.

Introduction

Préprocesseur

Fonctions principales d'un préprocesseur

- Les fonctions d'un préprocesseur varient selon le langage, mais elles incluent généralement :

3. Conditions de compilation :

Permet de compiler ou d'ignorer des parties du code selon des conditions définies.

Exemple :

Inclure ou exclure du code en fonction du système d'exploitation, par exemple :

```
#ifdef _WIN32
    printf("Code spécifique à Windows.\n");
#else
    printf("Code pour d'autres systèmes.\n");
#endif
```

Introduction

Préprocesseur

Fonctions principales d'un préprocesseur

- Les fonctions d'un préprocesseur varient selon le langage, mais elles incluent généralement :

4. Suppression de commentaires :

Certains préprocesseurs éliminent les commentaires pour simplifier le code avant de l'envoyer au compilateur.

Introduction

Préprocesseur

Illustration

Code Source Initial

```c
#include <stdio.h>
#define SQUARE(x) ((x) * (x))
/* Ceci est un commentaire*/
int main() {
    int a = 5;
    printf("Le carré de %d est %d\n", a, SQUARE(a));
    return 0;
}
```

Résultat du Préprocesseur :

1. Gestion des macros (#define SQUARE(x) ((x) * (x))) :
   - Toutes les occurrences de SQUARE(a) dans le code sont remplacées par ((a) * (a)).
2. Inclusion des fichiers d'en-tête (#include <stdio.h>) :
   - Le contenu du fichier d'en-tête stdio.h est inséré dans le fichier source.
3. Élimination des commentaires et optimisation :
   - Les commentaires sont supprimés.
   - Le code inutilisé est potentiellement éliminé (selon le cas).

Code Généré par le Préprocesseur :

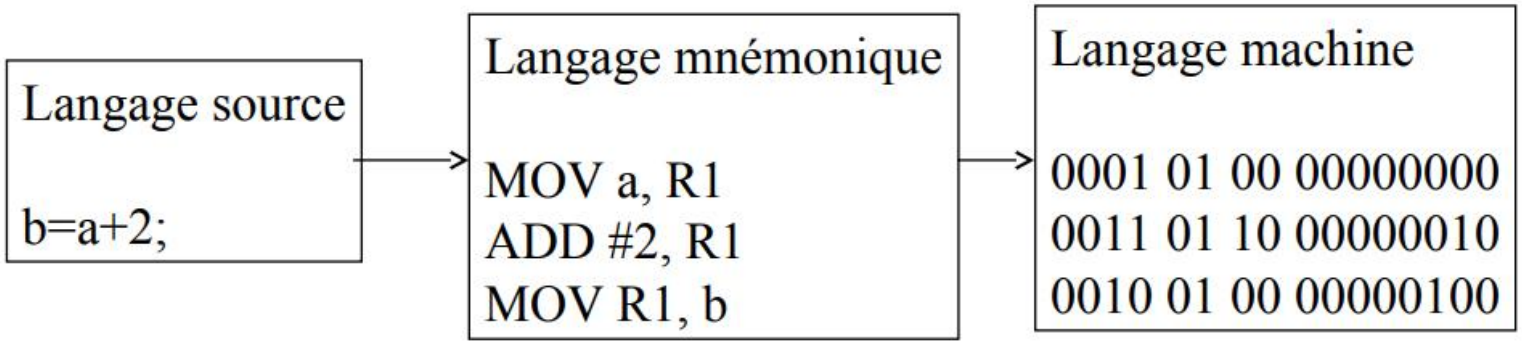
```c
/* Contenu du fichier stdio.h inséré ici */

int main() {
    int a = 5;
    printf("Le carré de %d est %d\n", a, ((a) * (a)));
    return 0;
}
```

Introduction

Assembleur

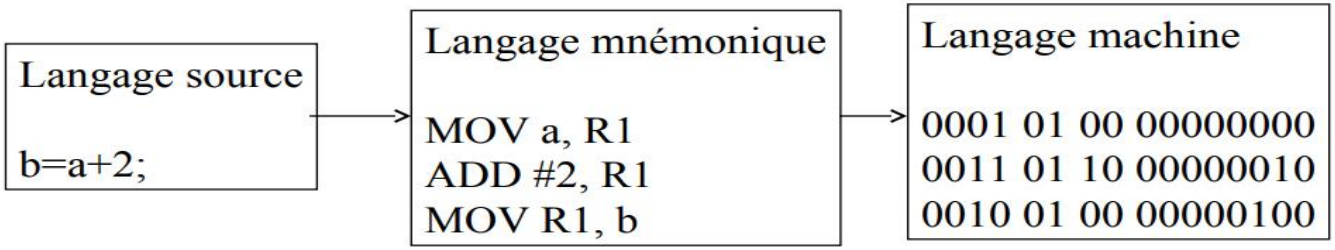
- Langage assembleur :
  - Langage de bas niveau, lisible par un humain.
  - Représente le langage machine à l'aide de **mnémoniques** (symboles faciles à retenir).
- Rôle de l'assembleur :
  - Convertit le code en langage assembleur en langage machine.
  - Produit un fichier exécutable prêt à être exécuté par le processeur.
- Processus avec un compilateur :
  - Certains compilateurs traduisent d'abord le code source en langage assembleur.
  - L'assembleur transforme ensuite ce code en langage machine.



Introduction

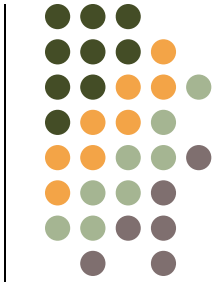
Assembleur

- Exemple



Traduction du Code Source en Code Machine

1. Langage Source :
   - Exprimé en langage de haut niveau (ex. b = a + 2;).
2. Langage Mnémotechnique (Assembleur) :
   - Traduit en instructions simples comme MOV, ADD, compréhensibles pour le processeur.
   - Exemple :
     - MOV a, R1 : Déplace a dans le registre R1.
     - ADD #2, R1 : Ajoute 2 au registre R1.
     - MOV R1, b : Déplace R1 dans b.
3. Langage Machine :
   - Instructions converties en code binaire directement exécutable par le processeur.
   - Exemple : MOV a, R1 → 0001 01 00 00000000.

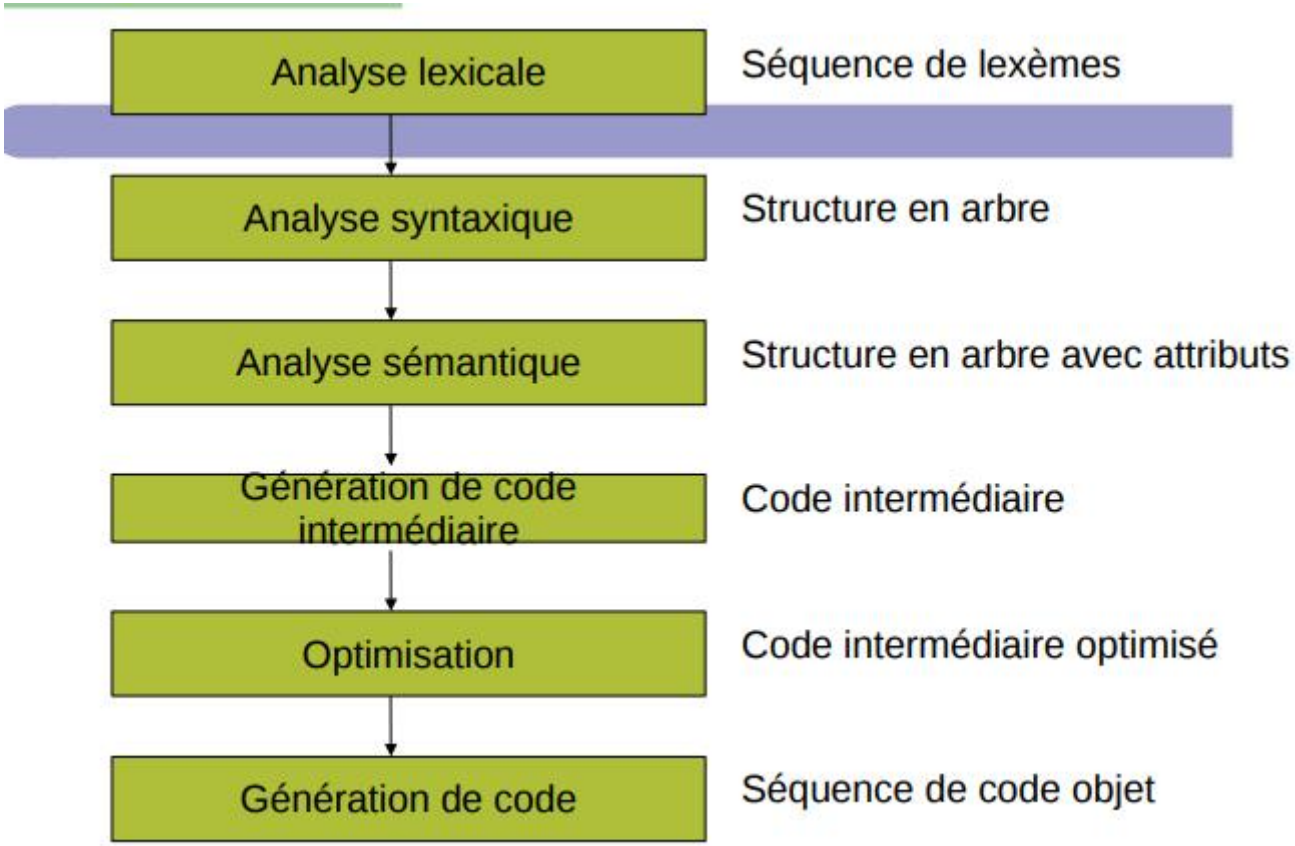


La compilation

Les phases de la compilation

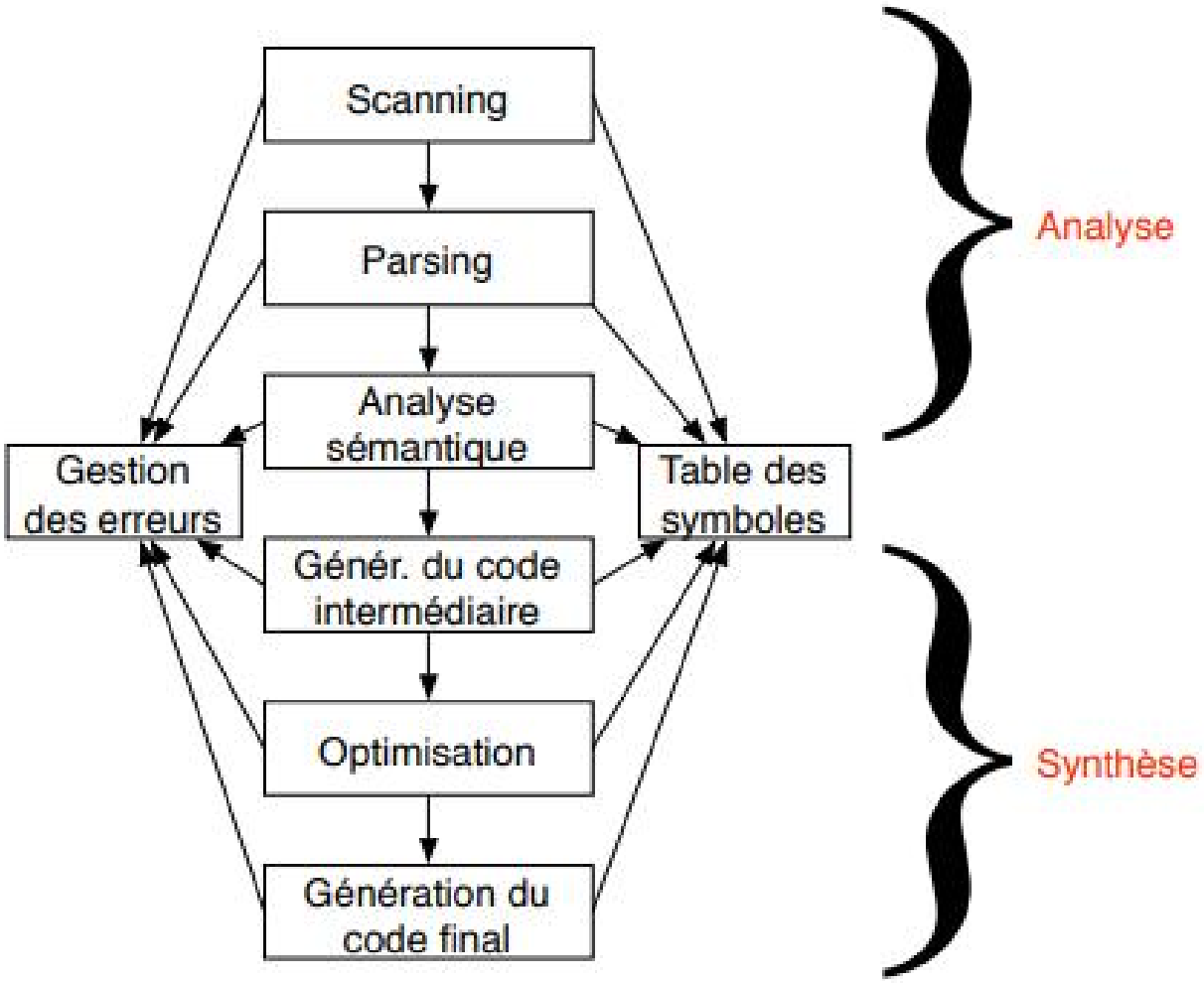
La compilation

Les phases de la compilation



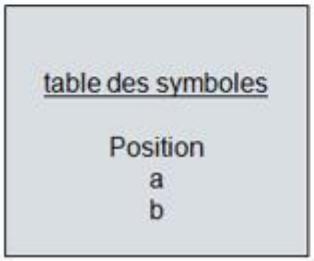
La compilation

Les étapes de la compilation



La compilation

Exemple

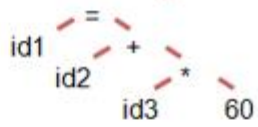


position = a+b*60

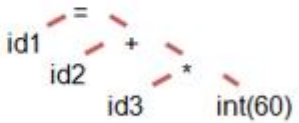
Analyseur lexical

id1 = id2 + id3*60

Analyseur syntaxique



Analyseur sémantique



Générateur de code intermédiaire

t1 = int(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3

Optimiseur de code

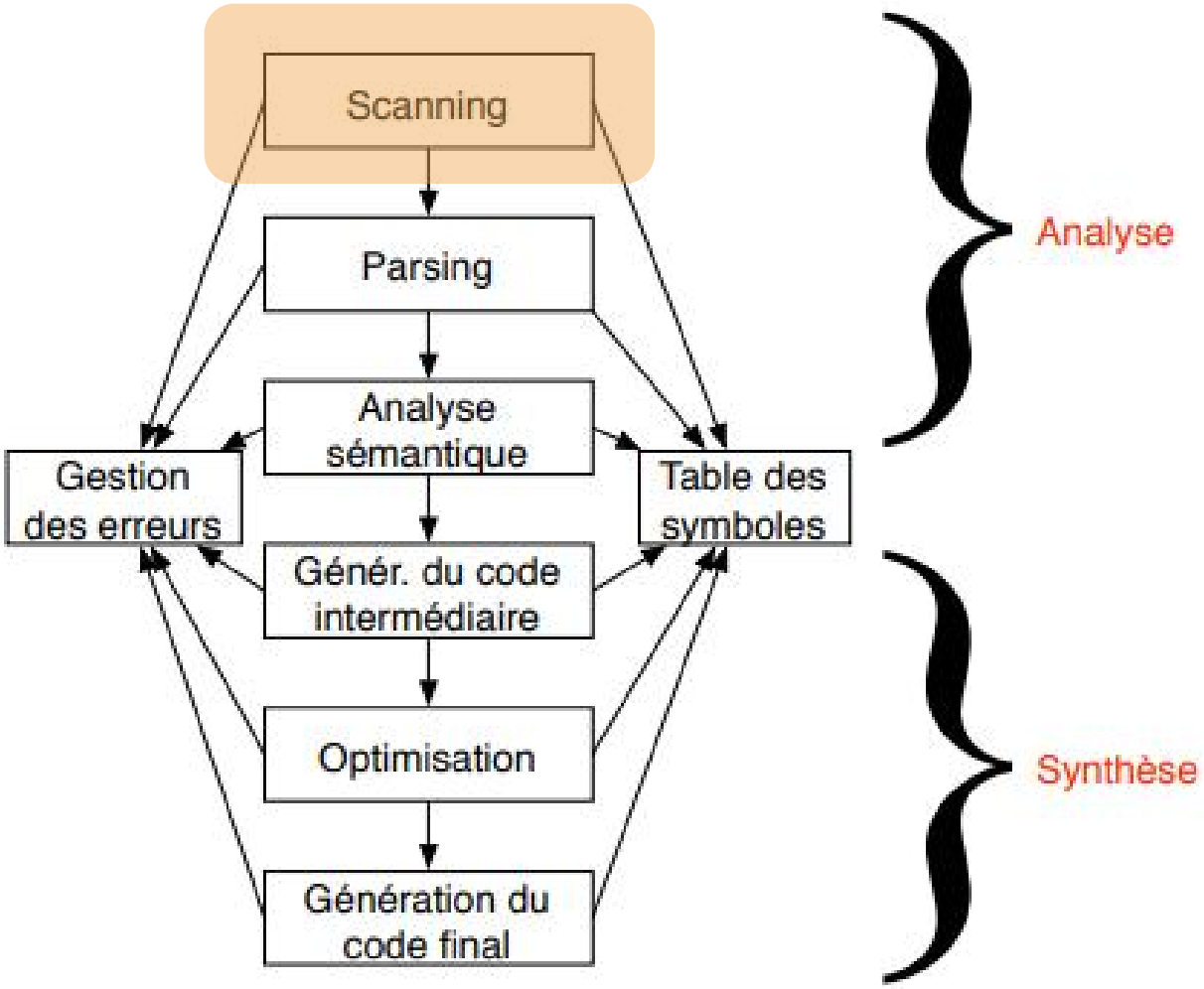
t1 = id3 * 60
id1 = id2 + t1

Générateur de code

Load r2,id3
Mul r2, #60
Load r1,id2
Add r1,r2
Store id1,r1

La compilation

Les étapes de la compilation



La compilation

Analyse lexicale (scanning)

- Un programme peut être vu comme une “phrase” ; le rôle principal de l’analyse lexicale est d’identifier les “mots” de la phrase.
- Le scanner décompose le programme en lexèmes en identifiant les unités lexicales de chaque lexème.

Exemple (de programme à compiler)

```
int main()
// Conjecture De Syracuse
// Hypothèse : N > 0
{
  long int N;

  cout << "Entrer Un Nombre : ";
  cin >> N;
  while (N != 1)
  {
    if (N%2 == 0)
      N = N/2;
    else
      N = 3*N+1;
  }
  cout << N << endl;  //Imprime 1
}
```

Exemple (de découpe en tokens)

```
int main ( )
// Conjecture De Syracuse
// Hypothèse : N > 0
{
  long int N ;

  cout << "Entrer Un Nombre : " ;
  cin >> N ;
  while ( N != 1 )
  {
    if ( N % 2 == 0 )
      N = N / 2 ;
    else
      N = 3 * N + 1 ;
  }
  cout << N << endl ; //Imprime 1
}
```

La compilation

Analyse lexicale (scanning)

Unité lexicale (ou Token) :

- **Définition :** Une unité lexicale est **un type générique** regroupant des éléments lexicaux ayant une fonction ou un rôle similaire dans un langage.
- **Par exemple :** identificateurs, opérateurs, mots-clés, etc.
- **Exemple :**
  - Pour **x = 5;** les unités lexicales sont :
    - identificateur (x),
    - opérateur d'affectation (=),
    - nombre entier (5),
    - point-virgule (;).

La compilation

Analyse lexicale (scanning)

Lexème (ou string) :

- **Définition :** Un lexème est une **occurrence concrète d'une unité lexicale** dans un code source. C'est la valeur textuelle spécifique associée au type générique de l'unité lexicale.
- **Exemple :** Si "identificateur" est l'unité lexicale, alors x ou y sont des lexèmes correspondant.
- **Relation :** Les lexèmes sont des instances spécifiques des unités lexicales.

La compilation

Analyse lexicale (scanning)

Modèle (ou Pattern) :

- **Définition :** Un modèle est une règle ou une expression régulière utilisée pour décrire une unité lexicale.
- **Utilisation :** Ces modèles servent à identifier automatiquement des lexèmes dans le code source.
- **Exemple :**
  - Pour un identificateur, le modèle pourrait être **[a-zA-Z][a-zA-Z0-9]*** (une lettre suivie d'éventuellement plusieurs lettres ou chiffres).
  - Pour un entier, le modèle pourrait être [0-9]+ (une ou plusieurs occurrences de chiffres).

La compilation

Analyse lexicale (scanning)

Exemple

Code source :

superficie = largeur * longueur / 2

UL :

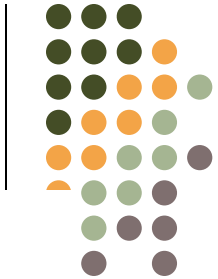
Id OpAff Id OpMult Id OpDiv Cste

Lexème :

superficie / largeur / longueur / 2

La compilation

Analyse lexicale (scanning): autres fonctionalités



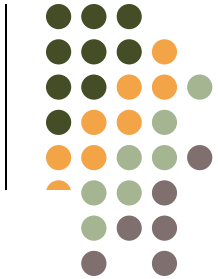
Ajout d'identificateurs et littéraux dans la table des symboles

- Pendant l'analyse lexicale, les identificateurs (par exemple, noms de variables, fonctions, etc.) et littéraux (par exemple, nombres, chaînes de caractères, etc.) peuvent être enregistrés dans une table des symboles.
- Cette table est une structure de données utilisée par le compilateur pour **associer chaque identificateur ou littéral avec des informations supplémentaires**, comme son type, sa portée, ou son emplacement en mémoire.

Note : Cet enregistrement peut également être effectué lors d'une phase ultérieure d'analyse (analyse syntaxique ou sémantique).

La compilation

Analyse lexicale (scanning): autres fonctionalités



Exemple :

Pour le code suivant :

```
int x = 5;
float y = 10.5;
```

- Les identificateurs x et y sont ajoutés à la table des symboles avec leurs types respectifs (int et float).
- Le littéral 5 et 10.5 peuvent aussi être ajoutés pour référence.

La compilation

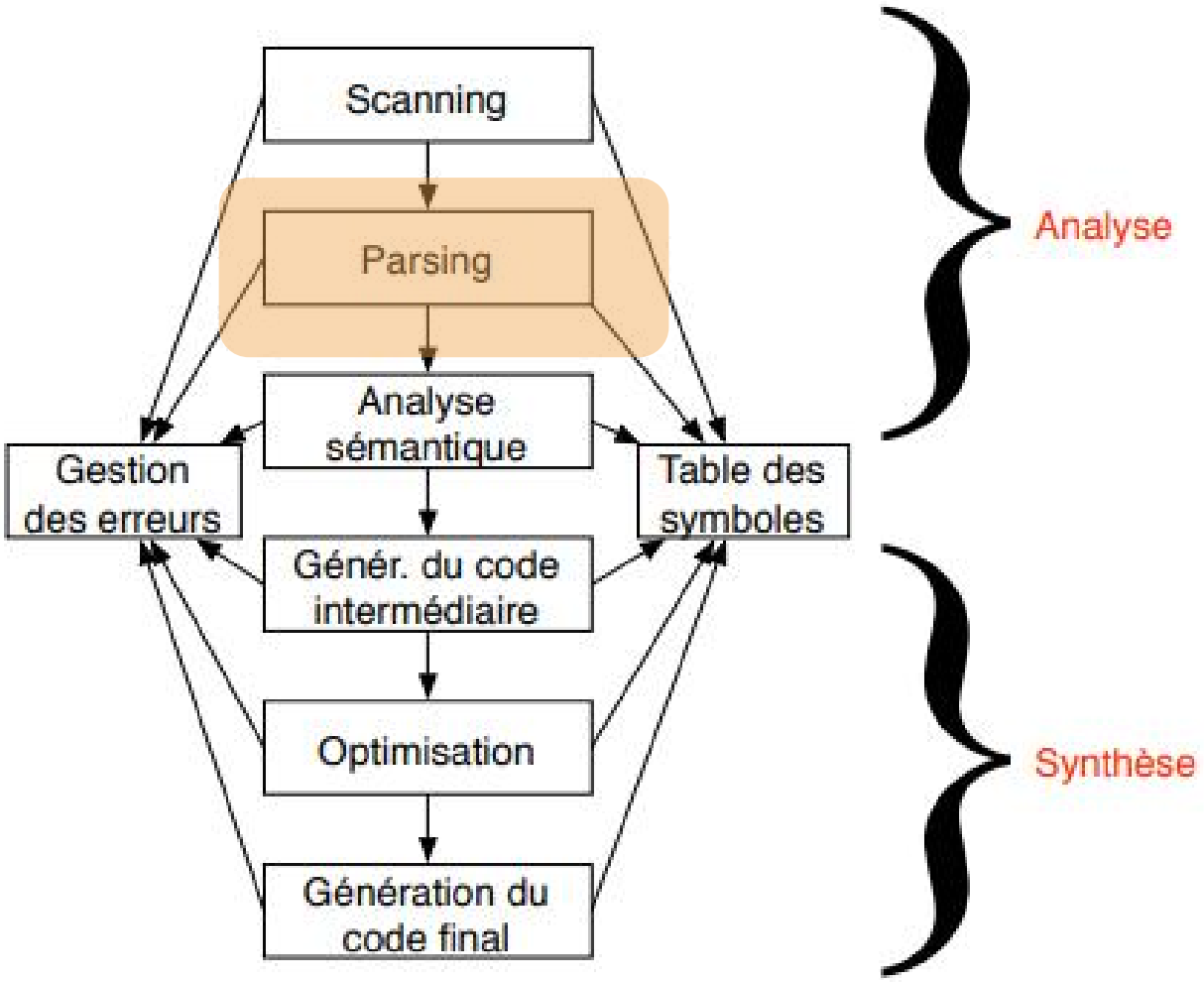
Analyse lexicale (scanning)

Production du listing/lien avec un éditeur intelligent

- Générer un listing (liste organisée) du code source analysé, souvent enrichi d'informations utiles pour l'utilisateur, comme les numéros de ligne ou les erreurs détectées.
- Cet objectif peut aussi être lié à l'intégration avec un éditeur de code intelligent (comme Visual Studio Code, Eclipse, etc.), permettant de :
  - Afficher les erreurs lexicales ou syntaxiques en temps réel.
  - Faciliter le lien entre le code source et les erreurs ou avertissements.

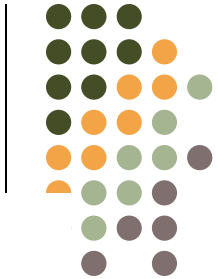
La compilation

Les étapes de la compilation



La compilation

Analyse syntaxique (parsing)



- **Validation des règles du langage**

L'analyse syntaxique vérifie que la suite d'unités lexicales respecte les règles définies par la grammaire du langage.

- **Organisation du programme**

Elle identifie la structure logique du code pour s'assurer qu'il est bien formé.

- **Construction de l'arbre syntaxique**

L'analyse produit un arbre qui représente la structure hiérarchique du code. Cet arbre décrit comment les éléments du programme sont liés entre eux.

- **Utilisation d'une grammaire formelle**

La grammaire hors-contexte est utilisée pour définir les structures syntaxiques autorisées dans le programme.

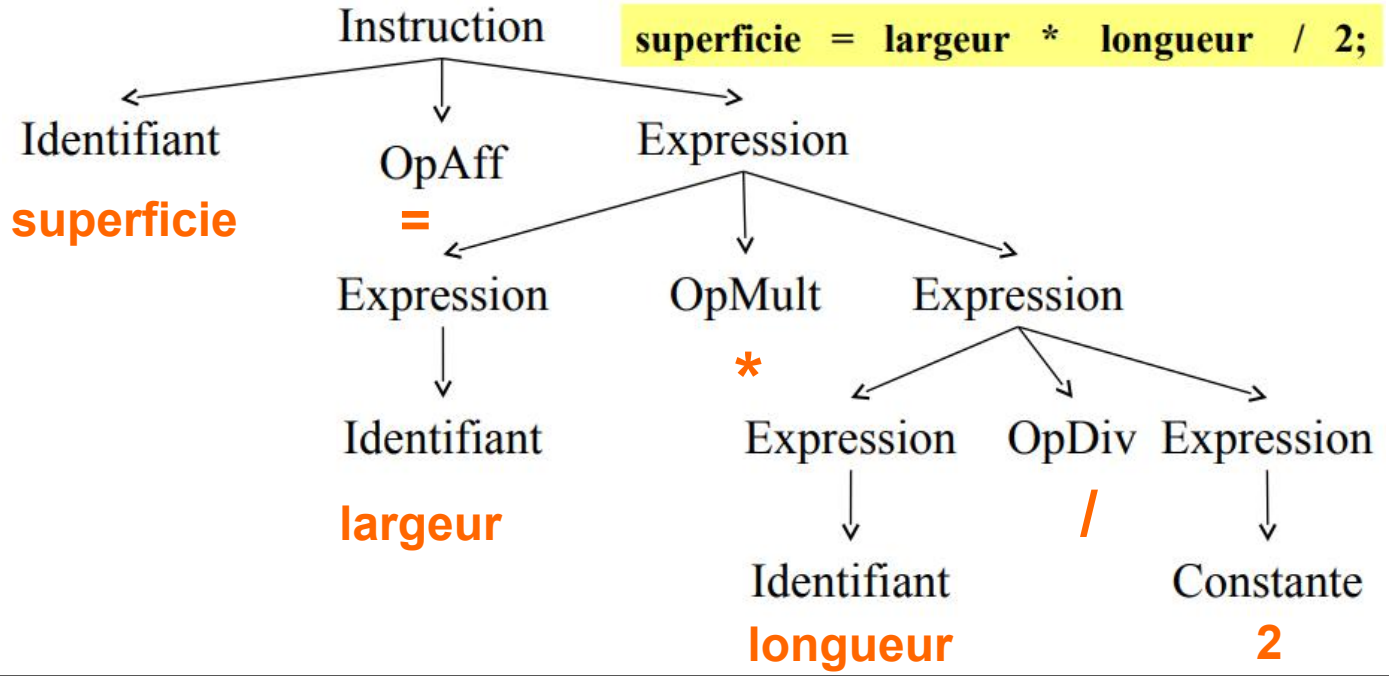
La compilation

Analyse syntaxique (parsing)

Exemple

Exemple de grammaire :

Expression ::= Identifiant
Expression ::= Constante
Expression ::= Expression OpMult Expression
Expression ::= Expression OpDiv Expression
Expression ::= Expression OpAdd Expression
Expression ::= Expression OpSub Expression
Instruction ::= Identifiant OpAff Expression



La compilation

Analyse syntaxique (parsing)

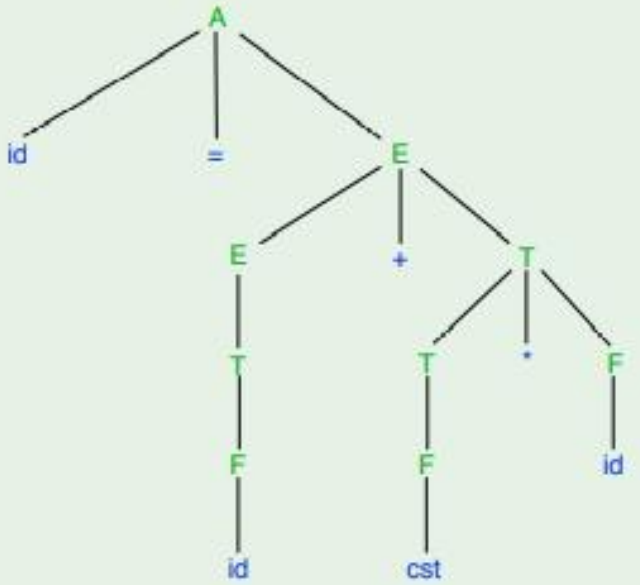
Exemple

Exemple (Grammaire d'une expression)

- A = "**id**" "**=**" E
- E = T | E "**+**" T
- T = F | T "*****" F
- F = "**id**" | "**cst**" | "**(**" E "**)**"

peut donner :

- **id = id**
- **id = id + cst * id**
- . . .



Arbre syntaxique de la phrase
id = id + cst * id

La compilation

Analyse syntaxique (parsing)

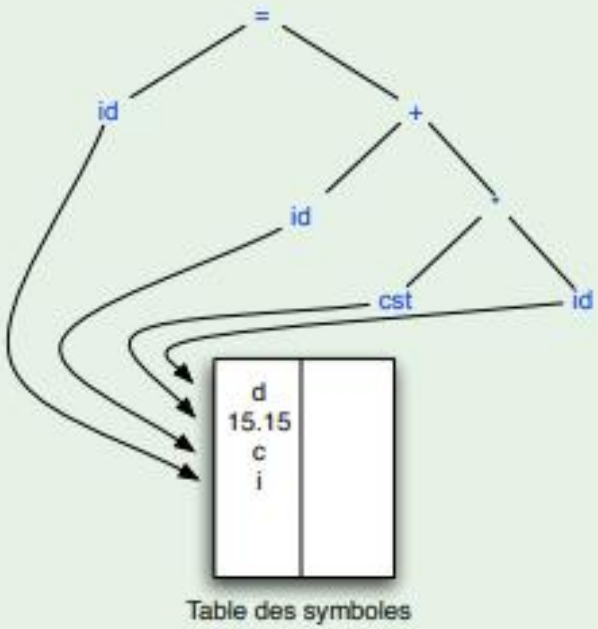
Exemple

Exemple (Grammaire d'une expression)

- A = **"id"** **"="** E
- E = T | E **"+"** T
- T = F | T **"*"** F
- F = **"id"** | **"cst"** | **"("** E **")"**

peut donner :

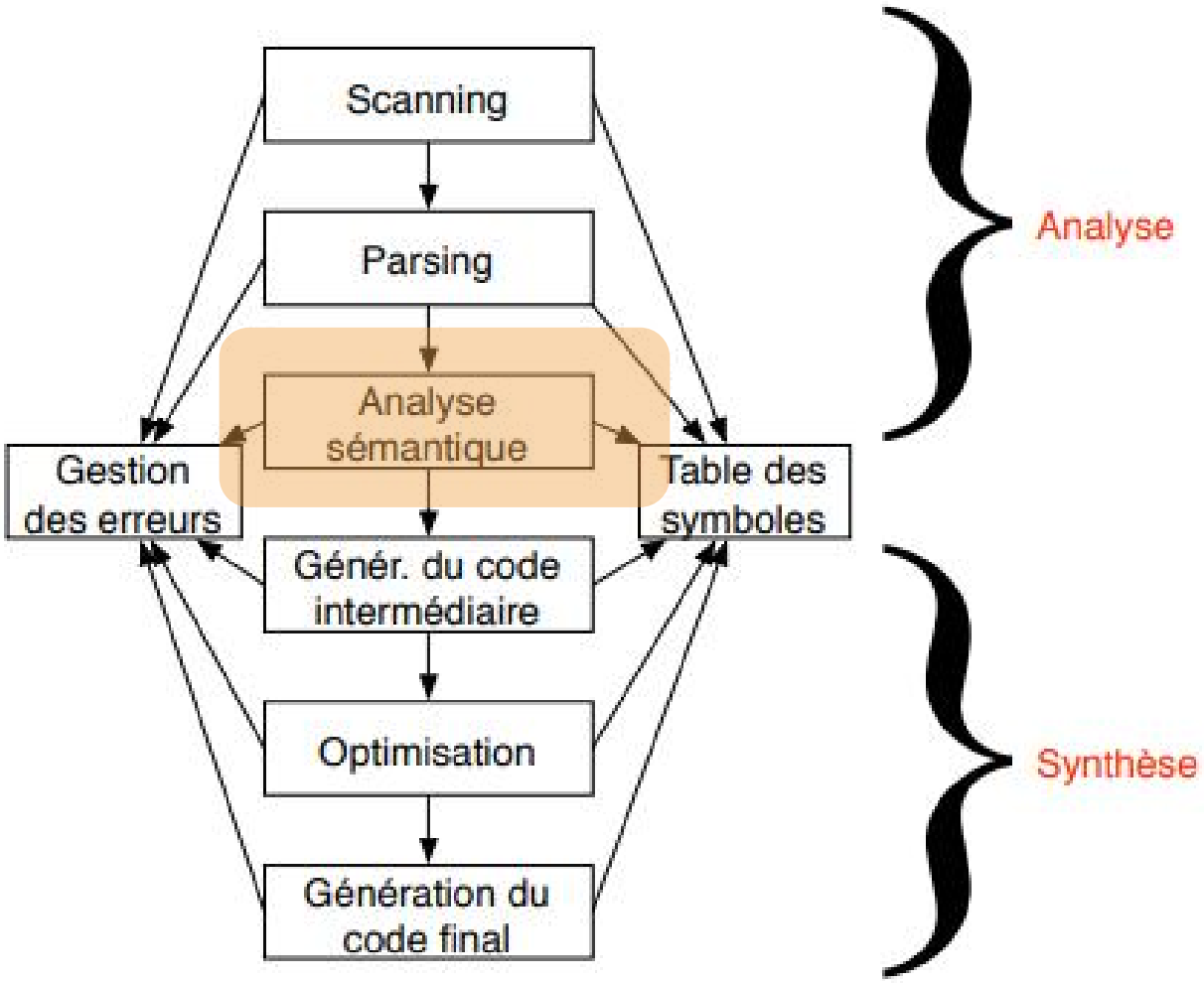
- **id = id**
- **id = id + cst * id**
- ...



Arbre syntaxique abstrait avec références à la table des symboles de la phrase **i = c + 15.15 * d**

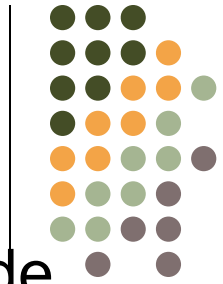
La compilation

Les étapes de la compilation



La compilation

Analyse sémantique



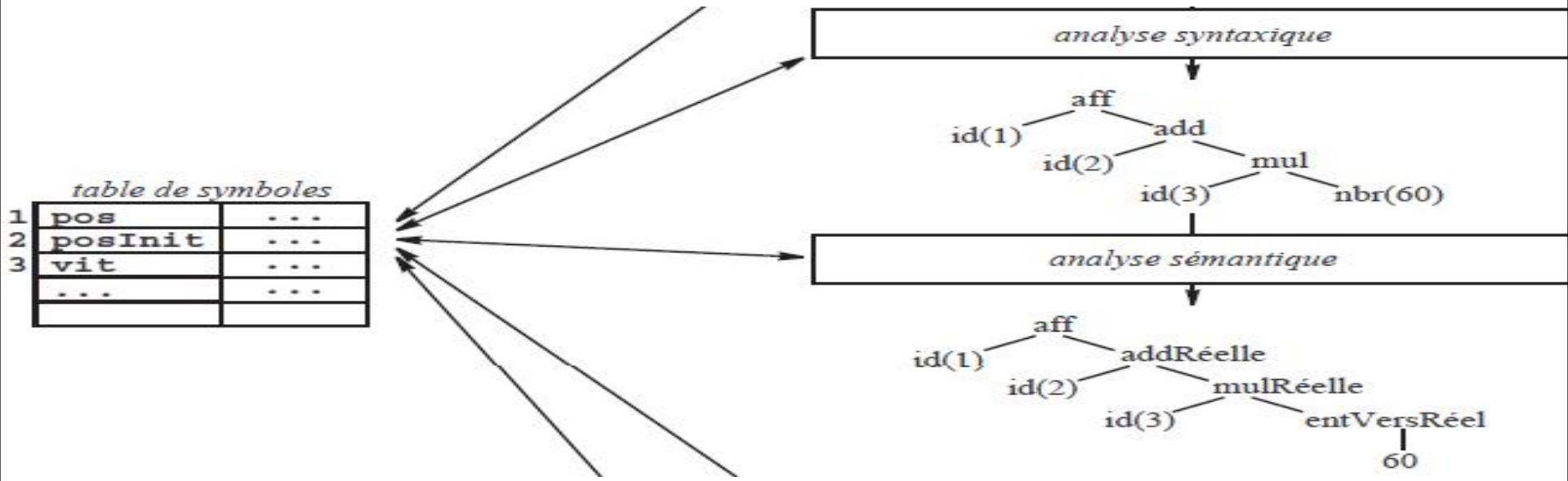
La structure du texte source étant correcte, il s'agit ici de vérifier certaines propriétés sémantiques, c'est-à-dire relatives à la signification de la phrase et de ses constituants :

- les identificateurs apparaissant dans les expressions ont-ils été déclarés ?
- les opérandes ont-ils les types requis par les opérateurs ?
- les opérandes sont-ils compatibles ? N'y a-t-il pas des conversions à insérer ?
- les arguments des appels de fonctions ont-ils le nombre et le type requis ?
- etc.

La compilation

Analyse sémantique

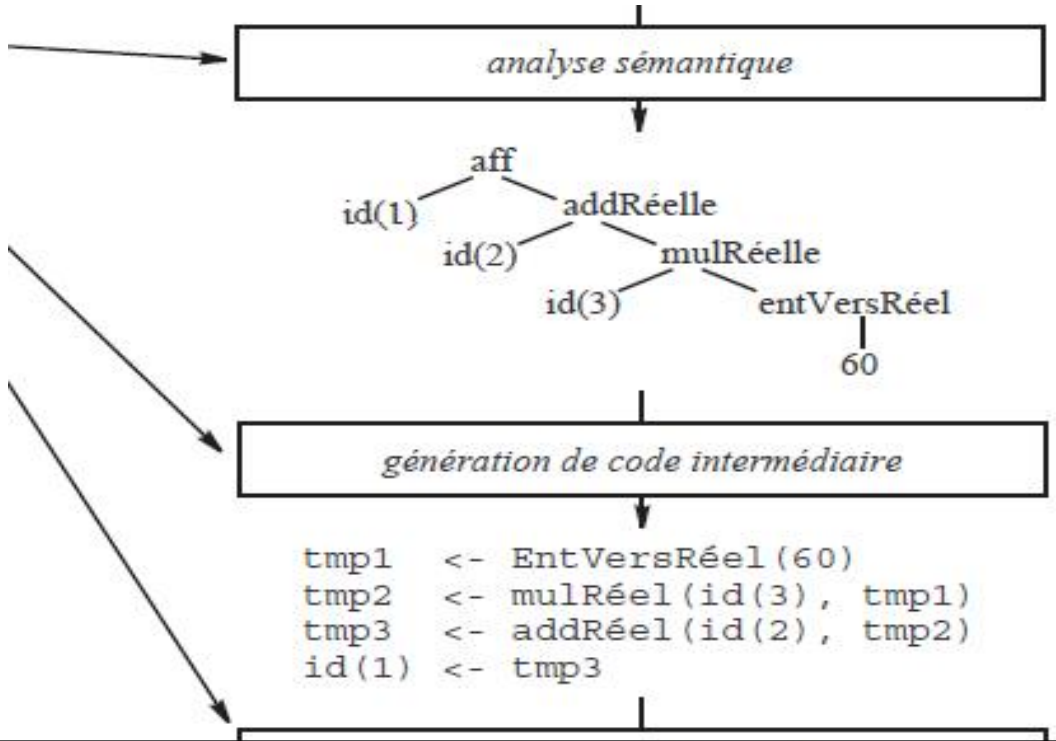
- Contrôle des règles sémantiques :
  - contrôle des types,
  - portée des identifiants
  - correspondance entre paramètres formels et paramètres effectifs
- Collecte des informations pour la production de code



La compilation

Génération de code intermédiaire

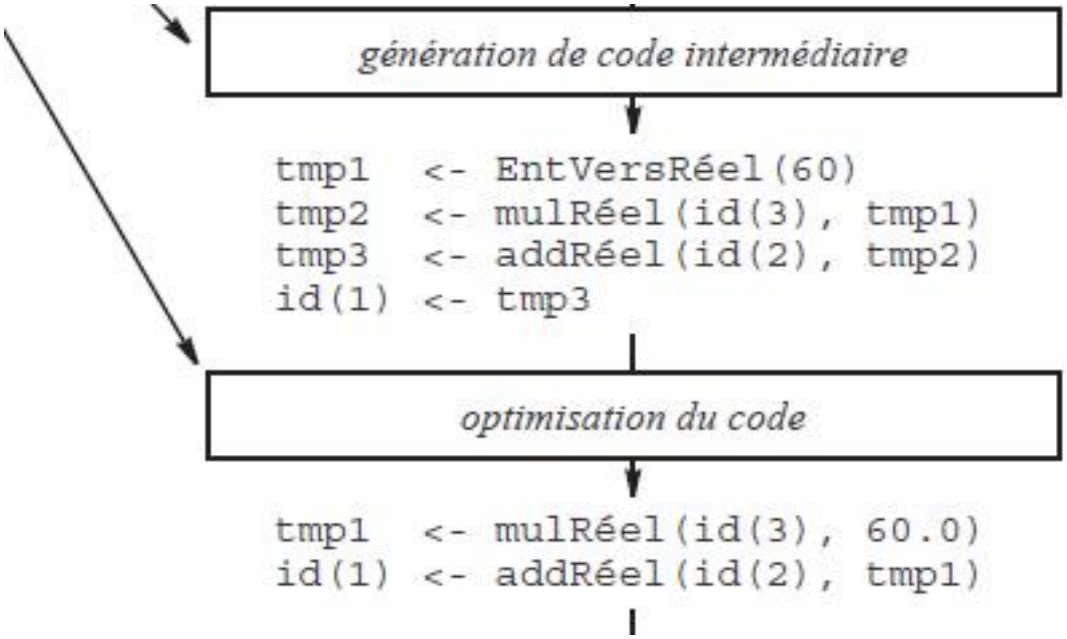
- Construction d'une représentation intermédiaire (pseudo-code) du programme source
- A partir de l'arbre sémantique, il est facile de produire du pseudo-code et de le traduire en langage cible



La compilation

Optimisation de code

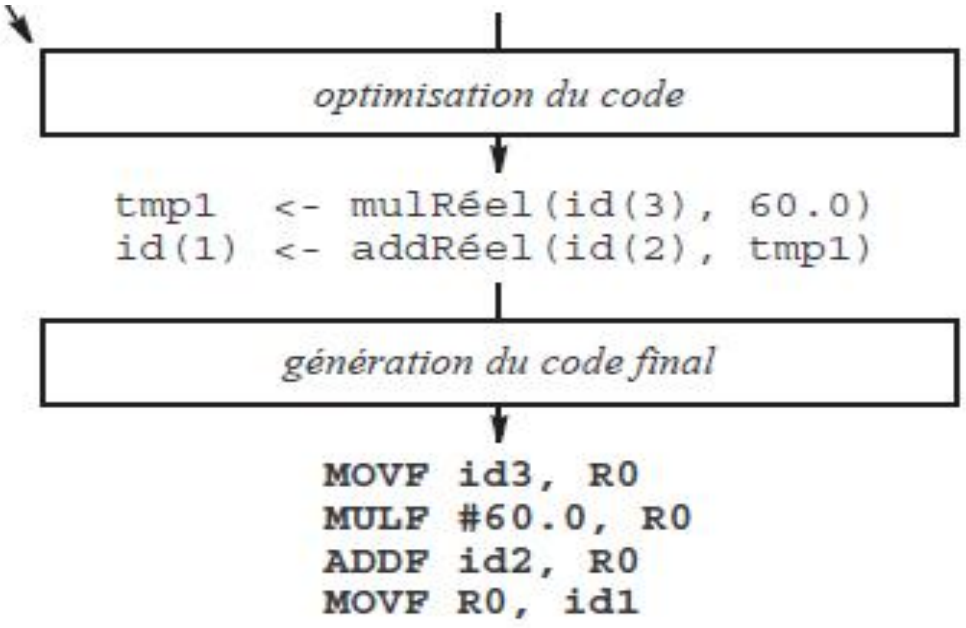
- Amélioration du code intermédiaire de façon que le code machine résultant s'exécute le plus rapidement possible



La compilation

Génération de code

- Production de code machine ou du code en langage d'assemblage à partir du code intermédiaire optimisé
- Sélection des emplacements mémoire pour chaque variable utilisées dans le programme
- Assignation des variables aux registres du processeur



La compilation

Exemple

Exemple (pour le code i = c + 15.15 * d)

1. Génération de code intermédiaire

```
temp1 <- 15.15
temp2 <- Int2Real(id3)
temp2 <- temp1 * temp2
temp3 <- id2
temp3 <- temp3 + temp2
id1   <- temp3
```

2. Optimisation du code

```
temp1 <- Int2Real(id3)
temp1 <- 15.15 * temp1
id1 <- id2 + temp1
```

3. Production du code final

```
MOVF   id3,R1
ITOR   R1
MULF   15.15,R1,R1
ADDF   id2,R1,R1
STO    R1,id1
```

La compilation

Gestion de la table des symboles

- Une **table des symboles** est une structure de données contenant un **enregistrement** pour chaque **identifiant**
- Quand **l'analyseur lexical** détecte un identifiant dans le programme source, cet identifiant est entré dans la table des symboles (s'il n'existe pas déjà)
- **L'analyse syntaxique** permet de compléter le **type** et la **portée** de la variable
- **L'analyse sémantique** consulte la table des symboles pour contrôler les types

La compilation

Gestion des erreurs

- Des erreurs peuvent être détectées aux différentes phases de la compilation
- Différents types d'erreurs :
  - **lexicale** (caractères inconnus, unité lexicale invalide…),
  - **syntaxique** (aucune règle syntaxique sélectionnée…),
  - **sémantique** (type incompatible…)
- Certaines erreurs ne doivent pas empêcher la génération de code (warning)

Merci pour votre attention!

Aida.lahouij@gmail.com