

PLP

Compilers

31 January 2026

Table des matières

1 Compilers	1
1.1 Frontend vs Backend	2
1.2 Compilation process	2
1.3 Compilation strategies	2
1.3.1 Ahead-of-time (AOT)	2
1.3.2 Just-in-time (JIT)	2
1.3.3 Transpilation	2
1.4 Intermediaire representations	3
1.4.1 Control flow graph	3
1.5 Compilation output	3
2 Abstract machines	3
2.1 Execution cycle	4
2.2 Turing machine	4
2.3 Stack machine	5
2.4 Virtual machine	5
2.5 Physical computers	5
3 Code generation	5
3.1 Target architecture	6
3.1.1 Instruction set	6
3.1.2 Instruction selection	6
3.2 Registers	6
3.2.1 Register allocation	7
3.3 Instruction scheduling	7
3.4 Code emission	7
4 Runtime environment	7
4.1 Memory management	8
4.2 Memory organization	8
4.3 Function call mechanisms	8
4.4 Stack and base pointers	9
4.4.1 Stack frames	9
4.4.2 Function prologue and epilogue	9
5 Code optimization	9
5.1 Constant folding	10
5.2 Common subexpression elimination	10
5.3 Dead code elimination	10
5.4 Constant propagation	10
5.5 Function inlining	11
5.6 Optimization levels	11

1 Compilers

Un compilateur est un programme prenant en entrée un programme et produisant en sortie un autre programme dans un code bas niveau (généralement du code machine).

1.1 Frontend vs Backend

Le **frontend** s'occupe de gérer l'analyse lexical, syntaxique et sémantique. Le **frontend** produit une représentation intermédiaire du programme qui sera utilisée par le **backend**.

Le **backend** consomme la représentation intermédiaire et produit le code machine. Le **backend** s'occupe aussi de traduire les constructions de haut niveau en constructions de bas niveau. Il s'occupe aussi de l'optimisation du code.

1.2 Compilation process

La compilation se fait en plusieurs étapes:

1. Analyse lexicale: transformation du code source en une suite de tokens.
2. Analyse syntaxique: transformation de la suite de tokens en un arbre syntaxique abstrait (AST).
3. Analyse sémantique: vérification de la validité sémantique du programme (types, portées, etc.).
4. Génération de la représentation intermédiaire (IR).
5. Optimisation de la représentation intermédiaire.
6. Génération du code machine à partir de la représentation intermédiaire optimisée.

1.3 Compilation strategies

1.3.1 Ahead-of-time (AOT)

Le code est compilé entièrement avant son exécution. Cela permet d'optimiser le code de manière plus agressive, mais peut entraîner des temps de compilation plus longs.

1.3.2 Just-in-time (JIT)

Le code est compilé au moment de son exécution. Cela permet d'adapter le code aux conditions d'exécution, mais peut introduire une latence lors de la première exécution.

1.3.3 Transpilation

Le code source est traduit d'un langage de haut niveau à un autre langage de haut niveau.

// ES6 JavaScript Code	// Transpiled ES5 JavaScript Code
<pre>const hello = (name) => { return 'Hello, \${name}! ' };</pre>	<pre>var hello = function (name) { return "Hello, " + name + "! " };</pre>
<pre>const person = { name: "John", age: 30, greet() { console.log(hello(this.name)); }, };</pre>	<pre>var person = { name: "John", age: 30, greet: function () { console.log(hello(this.name)); }, };</pre>
<pre>person.greet();</pre>	<pre>person.greet();</pre>

Fig. 1. – Capture des slides du cours – Exemple de transpilation

1.4 Intermediaire representations

L'interprétation intermédiaire (IR) est une représentation du programme qui se situe entre le code source et le code machine. Elle permet de faciliter l'optimisation et la génération de code. Il existe plusieurs types de représentations intermédiaires:

1. **Three-address code**: chaque instruction est représentée par une opération et jusqu'à trois opérandes.
2. **Static single assignment (SSA)**: chaque variable est assignée une seule fois, ce qui facilite l'analyse et l'optimisation.
3. **Control flow graph (CFG)**: représente le flux de contrôle du programme sous forme de graphe.

1.4.1 Control flow graph

Voici un exemple de graphe de flux de contrôle:

```
int fact(int n) {
    if (n <= 1) {
        return 1;
    }
    return n * fact(n - 1);
}
```

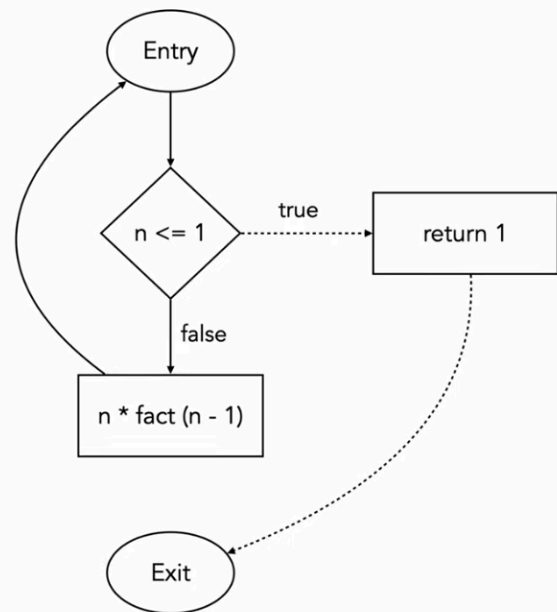


Fig. 2. – Capture des slides du cours – Exemple de Control Flow Graph

1.5 Compilation output

Il existe, de manière générale, trois types de sorties possibles pour un compilateur:

1. **Executable Binary**: le code machine est produit et peut être exécuté directement par le processeur.
2. **Intermediate Code**: le code est produit dans une représentation intermédiaire, qui sera ensuite interprétée ou compilée à la volée (par exemple: bytecode Java).
3. **Assembly Code**: le code est produit en langage d'assemblage, qui doit être assemblé en code machine par un assembleur.

2 Abstract machines

Les machines abstraites sont des modèles théoriques de machines qui permettent de comprendre le fonctionnement des compilateurs et des langages de programmation. Elles fournissent une couche d'abstraction entre le code source et le matériel. Ces machines prennent leur origine de la machine de Turing.

1. **Turing machines**: modèle théorique de calcul qui utilise une bande infinie et une tête de lecture/écriture.
2. **Stack machines**: modèle de machine qui utilise une pile pour stocker les opérandes et les résultats des opérations.
3. **Virtual machines**: modèle de machine qui exécute du bytecode, souvent utilisé pour les langages interprétés ou semi-compilés (comme la JVM pour Java).

2.1 Execution cycle

Le cycle d'exécution d'une machine abstraite comprend plusieurs étapes:

1. Fetch: récupération de l'instruction à exécuter.
2. Decode: décodage de l'instruction pour déterminer l'opération à effectuer.
3. Execute: exécution de l'instruction.
4. Memory access: accès à la mémoire si nécessaire (lecture/écriture).
5. Write back: écriture du résultat dans le registre ou la mémoire.

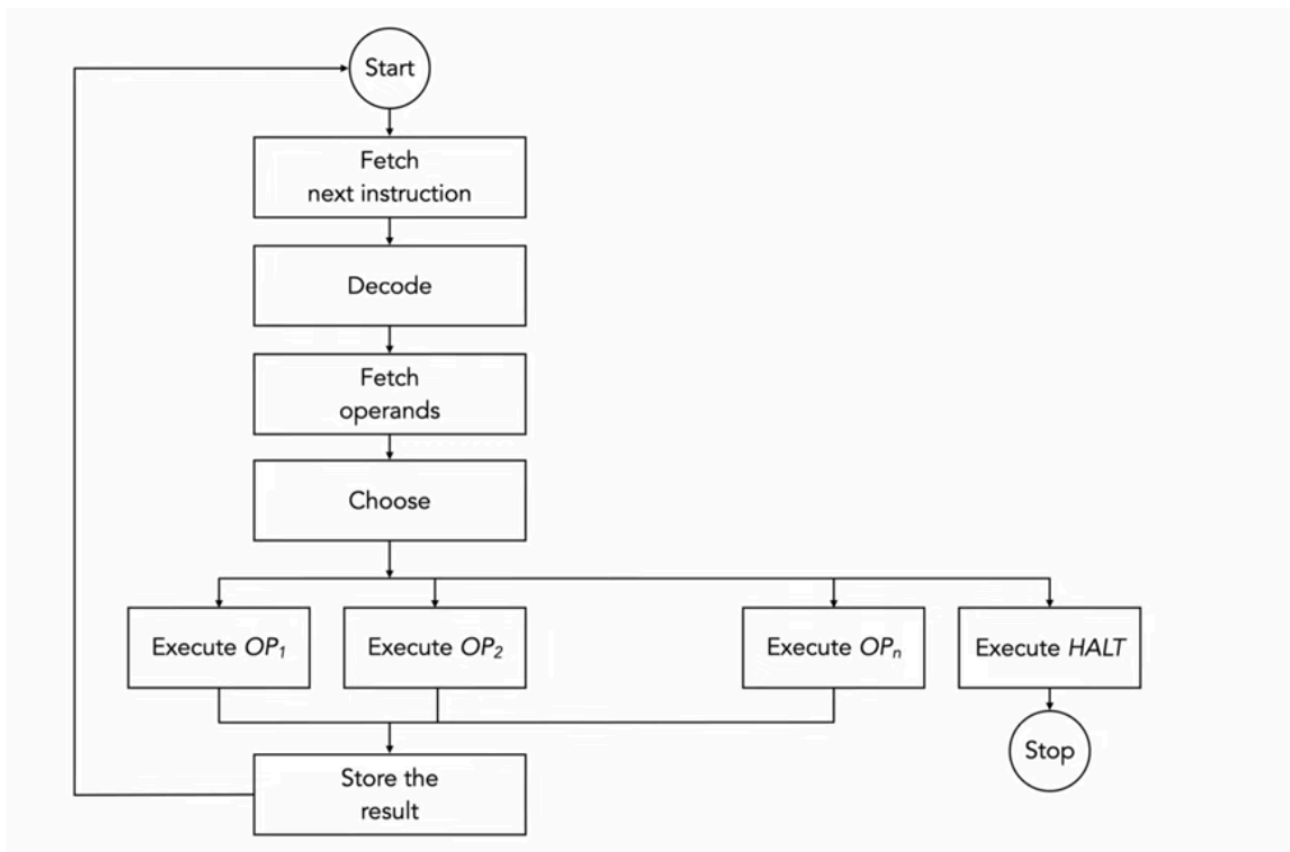


Fig. 3. – Capture des slides du cours – Cycle d'exécution d'une machine abstraite

2.2 Turing machine

Les machines de Turing ont été introduites par Alan Turing en 1936 pour formaliser le concept de calculabilité. Une machine de Turing est composée de:

1. Une bande infinie divisée en cellules, chacune pouvant contenir un symbole.
2. Une tête de lecture/écriture qui peut se déplacer le long de la bande.
3. Un ensemble d'états finis.

4. Une fonction de transition qui détermine l'action à effectuer en fonction de l'état actuel et du symbole lu.

2.3 Stack machine

Les machines à pile utilisent une pile pour stocker les opérandes et les résultats des opérations on parle souvent de **LIFO** (Last In First Out). Les instructions de la machine manipulent la pile en poussant et en retirant des valeurs. Les machines à piles sont souvent composée de:

1. **Instruction set**: ensemble d'instructions qui manipulent la pile (push, pop, add, sub, etc.).
2. **Memory model**: modèle de mémoire pour stocker les variables et les données.
3. **Execution model**: modèle d'exécution qui définit comment les instructions sont exécutées. Fetch, Decode, Execute, etc.
4. **Registers**: registres pour stocker les adresses et les valeurs temporaires.

2.4 Virtual machine

Les machines virtuelles sont des environnements d'exécution qui simulent une machine physique. Elles permettent d'exécuter du code dans un environnement isolé et contrôlé. Les machines virtuelles sont souvent utilisées pour:

1. Exécuter du bytecode (comme la JVM pour Java ou la CLR pour .NET).
2. Elles gèrent la mémoire, les exceptions et autres aspects de l'exécution du programme.

Elles sont particulièrement utilisées pour des langages interprétés ou semi-compilés.

2.5 Physical computers

Les ordinateurs physiques sont des machines réelles qui exécutent du code machine. Ils ont plusieurs fonctionnalités:

1. Ils opèrent directement sur le matériel en exécutant des instructions en code machine.
2. Ils offrent des performances optimales pour l'exécution des programmes.

3 Code generation

La génération de code est le processus de traduction de la représentation intermédiaire en code machine. Cela implique plusieurs étapes:

1. **Instruction selection:** sélection des instructions machine appropriées pour chaque opération dans la représentation intermédiaire.
2. **Register allocation:** allocation des registres pour stocker les variables temporaires et les résultats.
3. **Instruction scheduling:** ordonnancement des instructions pour optimiser l'utilisation des ressources du processeur.
4. **Code emission:** génération du code machine final.
5. **Linking and loading** (optionnel): liaison des différents modules de code et chargement en mémoire pour l'exécution.

3.1 Target architecture

La génération de code doit prendre en compte l'architecture cible pour laquelle le code est généré. Cela inclut:

1. **Performance Optimization:** optimisation du code pour tirer parti des caractéristiques spécifiques de l'architecture cible (par exemple, jeux d'instructions, pipeline, etc.).
2. **Memory Model:** gestion de la mémoire en fonction de l'architecture cible (par exemple, alignement des données, hiérarchie de cache, etc.).
3. **Instruction Set:** utilisation des instructions spécifiques à l'architecture cible.
4. **Endianness:** gestion de l'ordre des octets en fonction de l'architecture cible (big-endian vs little-endian).
5. **Platform Constraints:** prise en compte des contraintes spécifiques à la plateforme cible (par exemple, taille des registres, modes d'adressage, etc.).

3.1.1 Instruction set

Un ensemble d'instructions (Instruction Set Architecture - ISA) est un ensemble de commandes que le processeur peut exécuter. Chaque architecture de processeur a son propre ensemble d'instructions, qui définit les opérations de base que le processeur peut effectuer. De manière générale nous avons les types d'instructions suivants:

1. Instructions arithmétiques (ADD, SUB, MUL, DIV)
2. Instructions logiques (AND, OR, NOT, XOR, CMP, TEST)
3. Data transfer instructions (LOAD, STORE, MOVE)
4. Control flow instructions (JUMP, CALL, RETURN, BRANCH)

3.1.2 Instruction selection

La génération de code débute par la sélection des instructions appropriées pour chaque opération dans la représentation intermédiaire. Cela implique de mapper les opérations de haut niveau aux instructions machine spécifiques de l'architecture cible. Trois critères sont à prendre en compte:

1. Prupose: traduis correctement les opérations de la représentation intermédiaire.
2. Criteria for Selection: choisir les instructions qui minimisent le nombre d'instructions générées.
3. Techniques: utiliser du pattern matching pour identifier les séquences d'instructions optimales ou un approche table-driven pour mapper les opérations aux instructions.

3.2 Registers

Les registres sont des emplacements de stockage rapides situés à l'intérieur du processeur. Ils sont utilisés pour stocker des données temporaires et des adresses pendant l'exécution des programmes. La gestion efficace des registres est cruciale pour la performance du code généré. Il y a 2 types de registres:

1. **General-purpose registers:** utilisés pour stocker des données temporaires et des résultats des opérations.

2. **Special-purpose registers:** utilisés pour des fonctions spécifiques, comme le compteur de programme (PC) ou le registre d'état (FLAGS).

3.2.1 Register allocation

L'allocation des registres est le processus d'attribution des variables temporaires et des résultats aux registres disponibles dans le processeur. Cela implique de minimiser les accès à la mémoire en maximisant l'utilisation des registres.

3.3 Instruction scheduling

L'ordonnancement des instructions est le processus de réorganisation des instructions pour optimiser l'utilisation des ressources du processeur et minimiser les temps d'attente. Cela peut inclure:

1. **Data Dependencies:** gestion des dépendances de données entre les instructions pour éviter les conflits.
2. **Resource Contentions:** gestion des conflits d'accès aux ressources du processeur (unités fonctionnelles, bus, etc.).
3. **Pipeline Stalls:** minimisation des interruptions dans le pipeline d'exécution du processeur.

3.4 Code emission

L'émission de code est la dernière étape de la génération de code, où le code machine final est produit. Cela implique de convertir les instructions sélectionnées et ordonnancées en une séquence binaire exécutable par le processeur.

Le fichier de sortie peut-être un fichier exécutable binaire, un fichier d'assemblage ou un autre format spécifique à la plateforme cible.

1. **Binary format:** format binaire exécutable par le processeur.
2. **Assembly language:** code en langage d'assemblage qui doit être assemblé en code machine.
3. **Bytecode:** code intermédiaire qui sera interprété ou compilé à la volée.

4 Runtime environment

L'environnement d'exécution est l'ensemble des ressources et des services nécessaires pour exécuter un programme compilé. Cela inclut la gestion de la mémoire, les entrées/sorties, la gestion des exceptions, etc.

Les différentes responsabilités d'un environnement d'exécution sont:

1. Resource management
2. Memory handling
3. I/O operations
4. OS Communication
5. Function calls

4.1 Memory management

Un environnement d'exécution doit gérer la mémoire utilisée par le programme. Cela inclut:

1. **Memory allocation:** allocation de mémoire pour les variables et les structures de données.
2. **Memory deallocation:** libération de la mémoire lorsque celle-ci n'est plus nécessaire.
3. **Memory protection:** protection de la mémoire pour éviter les accès non autorisés.
4. **Garbage collection:** gestion automatique de la mémoire pour libérer les objets inutilisés.

4.2 Memory organization

L'organisation de la mémoire dans un environnement d'exécution comprend plusieurs segments:

1. **Text segment:** contient le code exécutable du programme.
2. **Data segment:** contient les variables globales et statiques.
3. **Heap:** zone de mémoire utilisée pour l'allocation dynamique.
4. **Stack:** zone de mémoire utilisée pour les appels de fonctions et les variables locales.

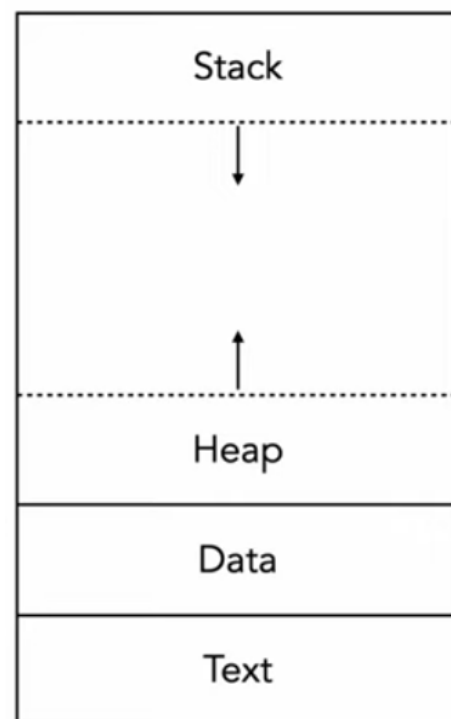


Fig. 4. – Capture des slides du cours – Organisation de la mémoire

4.3 Function call mechanisms

Les mécanismes d'appel de fonction définissent comment les fonctions sont appelées et comment les paramètres et les valeurs de retour sont gérés. Les principales étapes sont:

1. **Push parameters onto the stack:** les paramètres de la fonction sont poussés sur la pile.
2. **Transfer control to the called function:** le contrôle est transféré à la fonction appelée.

4.4 Stack and base pointers

La stack pointer (SP) et la base pointer (BP) sont des registres utilisés pour gérer la pile d'exécution. Le SP pointe vers le sommet de la pile, tandis que le BP pointe vers le début de la zone de la pile utilisée par la fonction en cours d'exécution. Ils sont utilisés pour accéder aux variables locales et aux paramètres de la fonction.

4.4.1 *Stack frames*

Un stack frame est une structure de données utilisée pour stocker les informations d'une fonction lors de son appel. Durant un appel de fonction:

1. La stack frame est créée et empilée sur la pile.
2. Les variables locales et les paramètres sont accédés via le base pointer.
3. À la fin de la fonction, la stack frame est dépilée et la mémoire est libérée.

4.4.2 *Function prologue and epilogue*

Le prologue et l'épilogue d'une fonction sont des séquences d'instructions qui sont exécutées au début et à la fin d'une fonction, respectivement. Le prologue prépare la pile et les registres pour l'exécution de la fonction, tandis que l'épilogue restaure l'état précédent avant de retourner au point d'appel.

Function Prolog:

1. Alloue de l'espace sur la pile pour les variables locales.
2. Sauvegarde les registres utilisés par la fonction.
3. Met à jour le base pointer.

Function Epilog:

1. Nettoie l'espace alloué sur la pile.
2. Restaure les registres sauvegardés.
3. Retourne au point d'appel.

5 Code optimization

D'un point de vue compilation, l'optimisation du code est le processus d'amélioration du code généré pour le rendre plus efficace en termes de performance et d'utilisation des ressources. Les optimisations peuvent être effectuées à différents niveaux, y compris au niveau de la représentation intermédiaire et au niveau du code machine.

Pour optimiser le code, l'objectif est de chercher à:

- Réduire le nombre d'instructions exécutées.
- Éliminer les parties redondantes ou inutiles du code
- Simplifier les expressions

Dans les techniques d'optimisation, nous retrouvons:

1. **Constant folding**: évaluation des expressions constantes à la compilation.
2. **Common subexpression elimination**: élimination des calculs redondants.
3. **Dead code elimination**: suppression du code qui n'est jamais exécuté.
4. **Constant propagation**: remplacement des variables constantes par leurs valeurs.
5. **Function inlining**: remplacement des appels de fonctions par le corps de la fonction.

5.1 Constant folding

Le **constant folding** est une technique d'optimisation qui consiste à évaluer les expressions constantes à la compilation plutôt qu'à l'exécution. Par exemple, l'expression `3 + 4` peut être remplacée par `7` directement dans le code généré. Surtout dans le cas où cette expression est utilisée plusieurs fois dans le code.

5.2 Common subexpression elimination

L'élimination des sous-expressions communes consiste à identifier les expressions qui sont calculées plusieurs fois et à les calculer une seule fois, en réutilisant le résultat. Par exemple,

```
a = b + c
d = b + c
peut être optimisé en:
temp = b + c
a = temp
d = temp
```

5.3 Dead code elimination

L'élimination du code mort consiste à supprimer les parties du code qui ne sont jamais exécutées. Par exemple, le code après une instruction de retour ou dans une branche conditionnelle qui n'est jamais prise peut être supprimé.

```
int f(int x) {
    int y = x + 1; // Unused variable
    return x * 2;
}
```

À la compilation, la variable `y` peut être supprimée car elle n'est jamais utilisée.

5.4 Constant propagation

La propagation des constantes consiste à remplacer les variables qui ont des valeurs constantes par leurs valeurs réelles. Par exemple,

```
x = 5
y = x + 2
peut être optimisé en:
y = 5 + 2
```

5.5 Function inlining

L'inlining de fonctions consiste à remplacer les appels de fonctions par le corps de la fonction elle-même. Cela peut réduire le surcoût des appels de fonctions, mais peut aussi augmenter la taille du code généré. Par exemple,

```
int add(int a, int b) {  
    return a + b;  
}  
  
int main() {  
    int result = add(3, 4);  
}
```

peut être optimisé en:

```
int main() {  
    int result = 3 + 4;  
}
```

5.6 Optimization levels

Les compilateurs offrent souvent différents niveaux d'optimisation qui permettent de contrôler l'agressivité des optimisations appliquées au code généré. Les niveaux d'optimisation courants sont:

1. **-O0**: pas d'optimisation, le code est généré tel quel.
2. **-O1**: optimisations de base qui n'affectent pas significativement le temps de compilation.
3. **-O2**: optimisations plus agressives qui améliorent la performance sans augmenter excessivement le temps de compilation.
4. **-O3**: optimisations très agressives qui peuvent augmenter le temps de compilation mais produisent le code le plus performant possible.