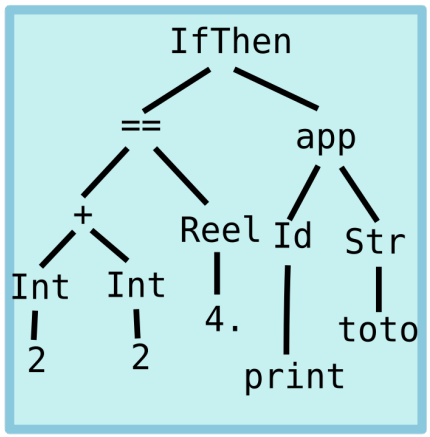
Rappel : Analyse Sémantique

Arbre syntaxique abstrait (AST)



Analyse sémentique

Analyse de portée

Quelle est la portée de chaque déclaration de variable et fonction

Dans les interpréteurs (et dans le projet) : Uniquement le "hosting"

ex : "Var" vs "Let" en JS

Typage statique

Pas dans le projet

Certains langages compilés ne sont pas statiquement typés

Liage statique

ex : méthodes statiques mais pas les méthodes d'instances

Pour chaque appel de fonction, si on sait statiquement quelle fonction est appellée, on y fait directement référence.

Pas dans le projet

L'analyse sémantique est un conscepte historique, dans les compilateurs modernes, il fait partie du backend (plus exactement des middlends)

Interpretation

Principe de l'Analyse Sémantique

On ne dit pas "une" analyse sémantique

Il s'agit d'une séquence d'analyses plus ou moins indépendantes. Chaque analyse parcours l'arbre sémantique pour récupérer de l'information.

Un AST en entrée mais aussi en sortie

On ne fait que analyser, manipuler, modifier, et enrichir cet AST.

Modification

- Enlever le sucre syntaxique,
- Analyse de portée,
- ...

Enrichissement

- Typage statique,
- liage statique des fonctions,
- ...

Ordre des analyses variables

Certaines analyses ont besoin qu'une autre se fasse avant, d'autres casserait une analyse que se fait donc après.

Enlever le sucre syntaxique

Langage minimaliste

Dans un backend complexe (avec des optim fines), on veut un langage réduit, car le code sera plus petit et plus sure.

Sucre syntaxique

Structures qui pourraient être remplacer par une autre plus verbeuse mais complètement équivalente.

Exemple : Les λ de Java sont des classes anonymes

si `expr` est de type `B`, alors `(A a => monExpr)` signifie :

```
new Function<A,B>(){
  public B apply (A a) {return monExpr;}
}
```

Remarque : c'est un choix d'encodage de l'ordre supérieur, on en verra un autre pour le projet.

Sucre syntaxique dans le projet

Les opérateurs booléens && et ||

$$e_1 \texttt{ \&\& } e_2 \quad \simeq \quad e_1 \texttt{ ? } e_2 \texttt{ : false}$$

$$e_1 \texttt{ || } e_2 \quad \simeq \quad e_1 \texttt{ ? true : } e_2$$

Pas cette année car l'opérateur ternaire _?_ : _ n'est pas demandé

Les boucles for

$$\texttt{for}(e_1; e_2; e_3)\, c \quad \simeq \quad e1; \texttt{ while}(e_2)\, \{\, c\ e_3; \,\}$$

Analyse de portée (et hoisting)

Shadowing

Un nom d'itentifiant (variable ou fonction) peut être réutilisé plusieurs fois. (ex : setters de java)

Idéal : déclaration ancêtre le plus proche

On remonte dans l'arbre et on trouve le dernier noeud de déclaration de cet identifiant.

Soucis 1 : les déclarations sont des feuilles

On modifie l'arbre.

Soucis 2 : Dans certains langages, on peut faire la déclaration plus tard

On fait remonter les déclarations (c'est le hoisting).

Exemples de hoisting en C

Depuis C99 on peut écrire

```
for (int i = 0; i++; i<3) {printf("%i",i);}
```

Exemples de hoisting en JS

En JavaScript on peut déclarer une variable bien plus tard

```
var i = 42;
print(f(i));
function f(j) {
  k = i+j
  var i = -20;
  return k;
}
```

Interlude : déclaration de variable JS

Variables déclarées

La déclaration est remontée en entête de la fonction courante.

Variables non-déclarés

Considérées comme déclarées tout au début du programme.

(comportement par défaut de la machine virtuelle du projet)

Et les fonctions ?

Comme les variables, sauf que leur "valeur" est remontée aussi.

Et les classes ?

Elles ne sont pas remontées (attendez la fin du cours si vous voulez me demander pourquoi)

Typage Statique

Typer tout ce que vous pouvez à la compilation

Différents algos :

- Sans inférence (Java) :
  l'utilisateur donne tous les types des variables et fonctions.
- Avec inférence (OCaml) :
  le compilateur les "devine", algo de complexité souvent horrible en théorie mais ok en pratique ("ordre" de vos fonctions peu élevé).

En général, seuls les expressions sont typées
(dans les langages comme OCaml où tout est expression)

Pas dans le projet

Car dans le projet les types sont dynamiques :
les types sont décidés pendant l'évaluation
Lent, lourd, mais plus "simple".

La vérification de type statique est trop lent pour les interpréteurs.

Liage statique de fonction

Fonction liée statiquement

Sur une application de fonction,

- déterminer statiquement quel code de fonction sera utilisé,
- ajouter, dans le noeud d'application, un pointeur vers cette fonction

(l'AST n'est alors plus un arbre)

Attention : pas toujours possible (ex : méthode dynamique, λ-expression)

Permet de faire des optimisations

- saut directement au bon endroit,
- inlining : remplacer la fonction par son code,
- ...

Projet : Pas de liage statique dans le projet.

Liage statique en Java

Liage statique total = méthodes statiques

Ne dépend pas de l'instance mais du type.

Liage dynamique = méthode non statique

Si le type est sous-classé, on ne sait pas quelle fonction sera appelée.

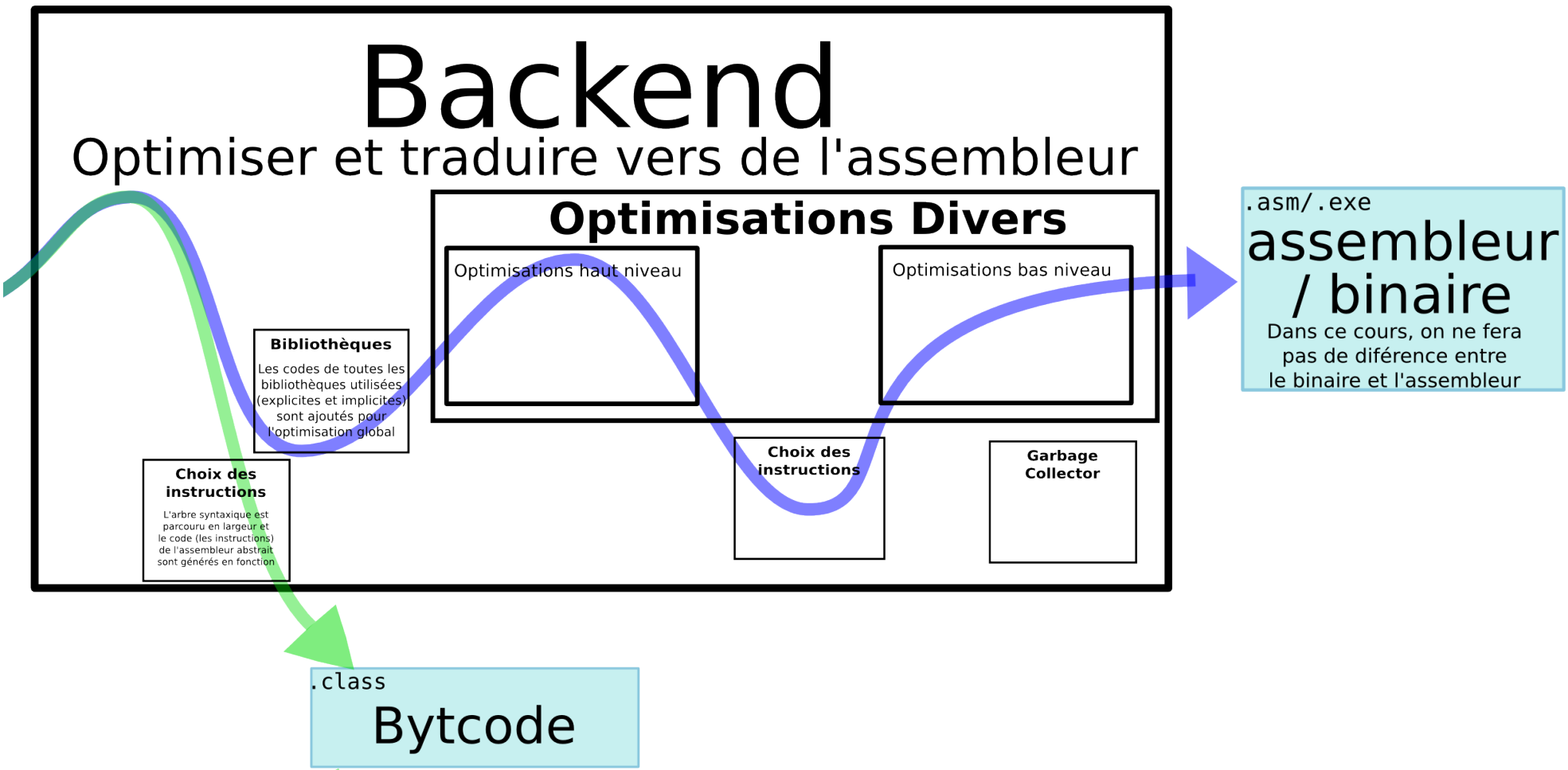
Liage statique partiel : surcharge de méthode

On ne sait pas quelle fonction est appliquée, mais on peut en restreindre le nombre à l'aide du type des arguments. On change donc les noms.

Exemple de liage en Java (extrait du TD de java dist.)

```
class A {
  public String f  (C obj) {return(" A et D ");}
  public String f  (A obj) {return(" A et A ");}}
class B extends A {
  public String f  (B obj) {return(" B et B ");}
  public String f  (A obj) {return(" B et A ");}}
class C extends B {}
class Test {
  public static void main ( String [] args ){
    A a1 = new A ();  A a2 = new B ();
    B b  = new B ();  C c  = new C ();
    System.out.println (a1.f  (a2));
    System.out.println (a1.f  (b));
    System.out.println (a1.f  (c));
    System.out.println (a2.f  (a2));
    System.out.println (a2.f  (b));
    System.out.println (a2.f  (c));
    System.out.println ( b.f  (a2));
    System.out.println ( b.f  (b));
    System.out.println ( b.f  (c));
```

Rappel : Backend



Dans le projet : Uniquement la flèche verte (vers le bas)

Assembleur/assemblé/objet/binaire

Binaire .exe	Object .o,.obj	Assemblé	Assembleur .asm, .S
Langage machine 1 block mémoire = 1 instruction	Binaire + liens externes	traduction littéral du code objet	Assemblé + macros + labels + psedo-instr.
0100111111111111	0100111111111111	JSR -1	loop : JSR loop

Cible du compilateur : le code objet

Dans le projet, on utilise plutôt le langage assemblé (plus lisible)

Pour simplifier on parle toujours d'assembleur...

Et on ne parle pas de prog. embarqué ou sur GPU

“Linking” de bibliothèques

Problème : joindre différents codes

Bibliothèques, projet, fonction C, système...

Trois solutions :

Compil. globale
- dans le backend
- + opti.

Compil. séparée
- objet→binaire
- compli. rapide

Appels systèmes
- à l'exec.
- droits...

En pratique on mixe les trois.

Optimisations

De nombreuses passes d'optimisation

Toutes avec des méthodes très différentes les unes de autres.
Certaines passes sont faites plusieurs fois.

gcc : près de 100 passes différentes !

Haut-niveau

Sur l'AST
analyse global,
fonctions/classes

gcc : 40+ passes

Call-graph

sur le flow-chart
boucles,
domaine des variables

gcc : 20+ passes

Bas-niveau

sur l'assembleur
choix d'instr.
ordre des instr.

gcc : 20+ passes

Peut être un cours de culture G sur une ou deux méthodes d'optimisation
En attendant voir le cours confiné de l'ENS lyon :
`https://www.youtube.com/playlist?list=PLtjm-n_Ts-J-6EU1WfVIWLhl1BUUR-Sqm`

Garbage Collector (GC)

En français : ramasse miette ou Glaneur de Cellules

Objectif

Libérer automatiquement les zones mémoires inutilisées

Thread indépendant

Mini-programme tournant sur un thread indépendant.

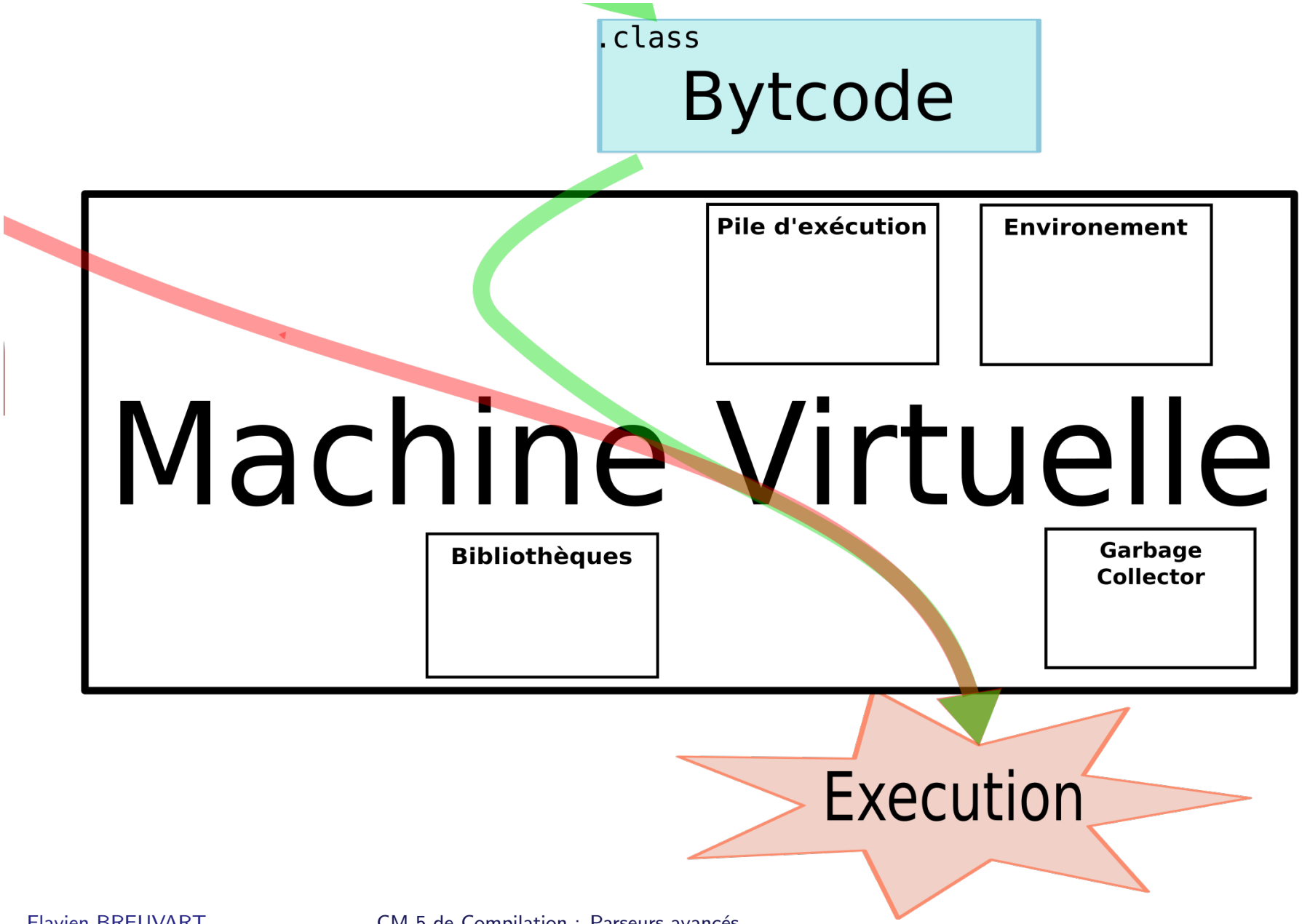
Méthode : libère les noeuds orphelins

càd les objets non accessibles (en suivant des pointeurs) depuis l'état courant (pile+cont.+registre)

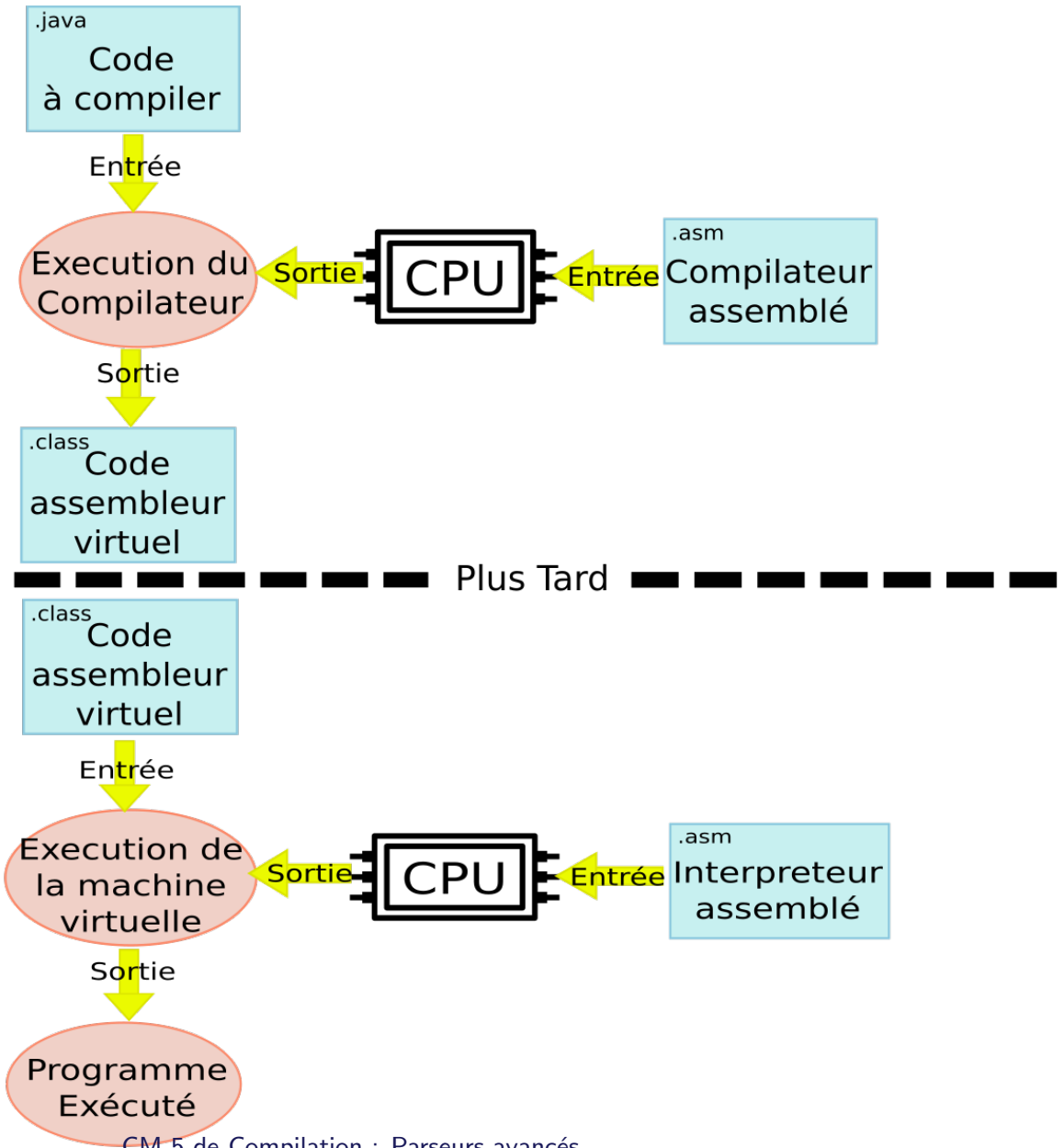
(Il y a d'autres méthodes moins répandues)

Peut être un cours de culture G sur les entrailles du GC.
En attendant la page wikipedia française est très bien

Rappel : Machine virtuelle



Rappel : Fonction de la VM



Qu'est-ce qu'une "machine" ?

Exécute immédiatement un programme déjà parsé

On peut aussi prendre de l'assembleur ou du bytecode (parseur trivial).

Quelques exemples de machines

Machine de Turing universelle (sur un encodage de Turing)
Machine virtuelle Java (sur du bytecode java)
Votre processeur (sur du bytecode machine)
Machine virtuelle JS (sur du JS dans l'interprèteur)
Machine de Krivine (sur du lambda-calcul)
Machine Mini-JS (sur l'assembleur vu en cours)
~~Machine virtuelle linux~~

machine : physique/virtuelle/abstraite

Physique	Virtuel	Abstraite
Hardwear →un processeur	Softwear →Java/JS/Python	Non-implémenté →M. de Turing
Efficace	**Adapté**	**Théorique**
mais complexe	et universelle	sémantique
structures	**structures**	**structures**
registres, tas, adressage, caches..	pile, contexte, objets, GC	Réécriture comme VM haut-niveau

Couches d'abstractions

...	...
psedo-code	algo
langage	programme
machine virtuelle/abstraite	programme/assembleur/binaire
système	assembleur
modèle de processeur	binaire assemblé
processeur	binaire + caches + ...
circuits séquentiel (horloge)	exécutions
...	...

Remarques :

- en C pas de machine virtuelle/abstraite,
- des fois des boucles :
  - bibliothèques abtraites par une interface puis utilisées,
  - machine virtuelle elle même exécutée par un langage,
  - psedo-instructions dans le processeur

La mini-JS machine

Machine virtuelle jouet utilisée en cours

Entrée : Assembleur mini-JS

Efficacité non considérée,

choix de design pédagogiques
(difficulté progressive)

je l'ai conçue et écrite, il peut donc y avoir des bugs

Assembleur Mini-JS

Assembleur adapté au JS :

- valeurs correspondant aux types de JS,
- comportements par défauts compatibles avec JS,
- instructions bien choisies.

Petit topo sur les éléments clés des machines (Physiques, virtuelles et abstraites)

Une structure relativement semblable d'une machine à l'autre

Zone de code, pile d'appels, valeurs entières, ect..

Des structures spécifiques aux machines physiques

Registres, manipulation des caches...

Des structures spécifiques aux machines virtuelles

Code en lecture seule, contexte courant...

Des structures emprunt d'un paradigme

Objets, continuations,...

La zone de code

Le code exécuté est disponible dans une zone mémoire au lancement de la machine

Séparée de la mémoire ? droit d'écriture ?

Les machine virtuelles séparent généralement données et code, ce dernier étant en lecture seul.

Code "autocompilant"

Il est possible, sur une machine physique d'écrire du code se modifiant lui-même. Utilisé pour offusquer le code (virus et copyright), mais aussi, plus récemment, pour recompiler plus efficacement

Réduction par réécriture

Certaines machines abstraites, et plus récemment virtuelles (Web-assembly) utilisent de la réduction du code à la manière du λ-calcul.

Registres vs Pile

Cible des instructions assembleurs

Machines physiques : les instructions manipulent les registres
Machines virtuelles : les instructions manipulent la pile

On ne peut pas se passer de la pile d'appel

Une opération arithmétique formant un arbre binaire équilibré de hauteur h composé de sommes flottantes a besoin de h registres.

Les registres ne sont qu'un cache "de niveau 0"

que l'on manipule "à la main".

Et qui n'introduit que des difficultés supplémentaires pour le programmeur.

(Dés)Alocation mémoire : problématique système

Récupération de la mémoire

Machines physiques : dans le tas (si non-plein) ou via le système (mémoire virtuelle)
Machines virtuelles/abstraites : idem mais caché...

Libération de la mémoire

Machines physiques : à la main (free)
Machines virtuelles/abstraites : via un GC embarqué (généralement celui du langage d'écriture de la VM)

La gestion du contexte

Qu'est-ce qu'un contexte ?

C'est le dictionnaire qui lie le nom des variables courantes à leur valeurs

Sur machines physiques

Pas de contexte explicite.

Encodé à la compilation (dans la pile ou une structure séparée).

Sur machines virtuelles

Deux choix :

- Table de hashage : très efficace,
- Liste chaînée : shadowing, copie.

En vrais :

- Mix des deux,
- + liage statique.

Les structs

Types de données

Machines physiques : tout est entier (sauf la lisp-machine)
Machines virtuelles : Typage dynamique, structure préconstruites

Quelques structures préconstruites

- types de bases (int, floats, ect...)
- objets
- tableau/string
- clôture/continuation (cf cours dédié)
- ...

Miscelaneous : Méta-programmation

Méta-opération dans un langage interprété

Manipuler la structure interne de la machine virtuelle dans le programme interprété.

Exemples

- Java : création dynamique d'une classe,
- JS : introspection du code d'une fonction (pas dans le projet)
- Python : modification dynamique du contexte courant.

Permet de faire de la magie, d'être efficace, mais est quasi-impossible à compiler, et pause des soucis de sûreté et sécurité.