Compilation

Florent Madelaine
florent.madelaine@unicaen.fr

Langages et compilation — L3 informatique

Aujourd'hui

- Introduction à la compilation
- Découverte d'une machine virtuelle et à son langage qui servira de langage cible pour le DM de compilation.

Première partie I

Du source à l'exécutable

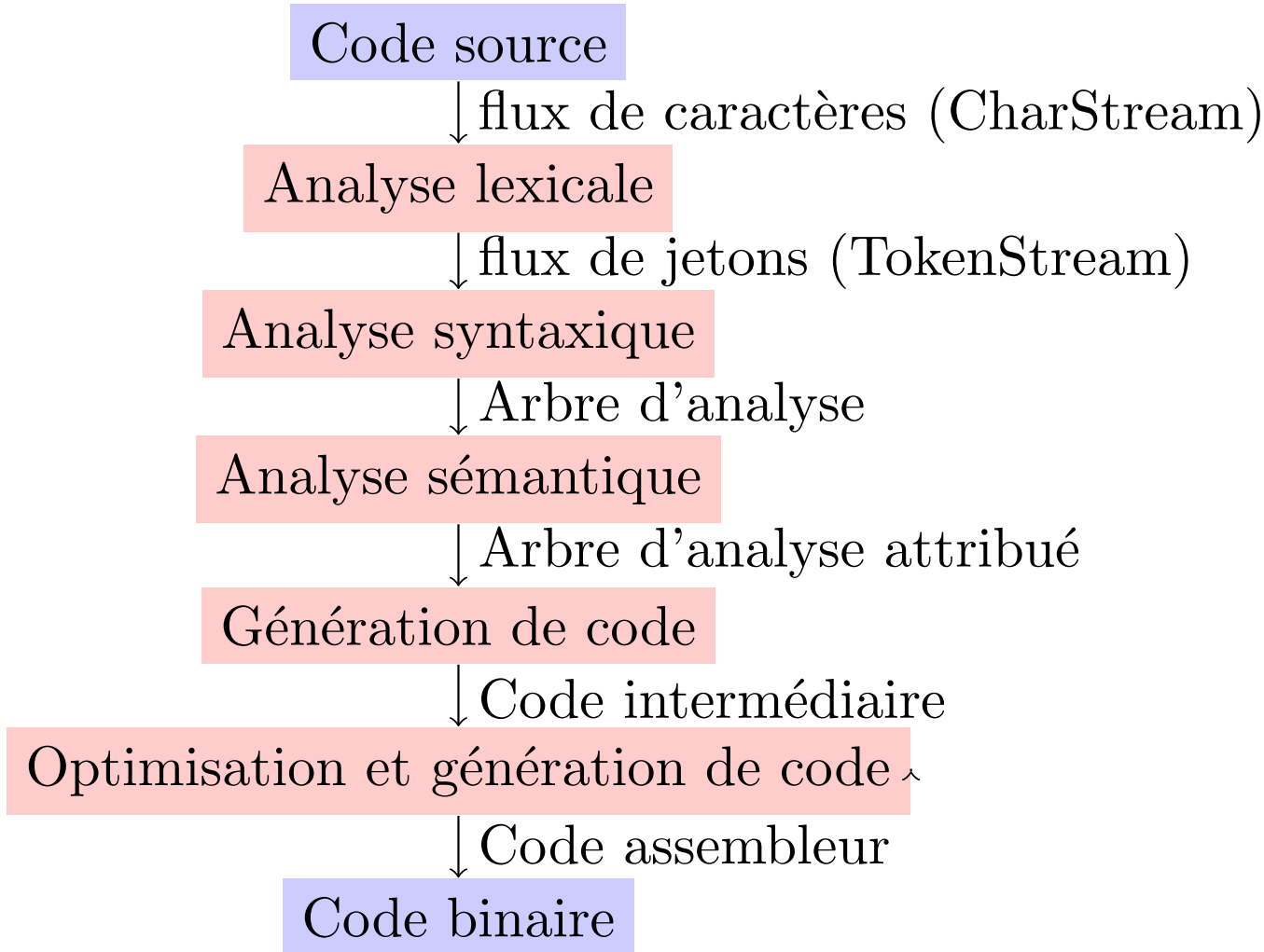
1. Introduction

Définition

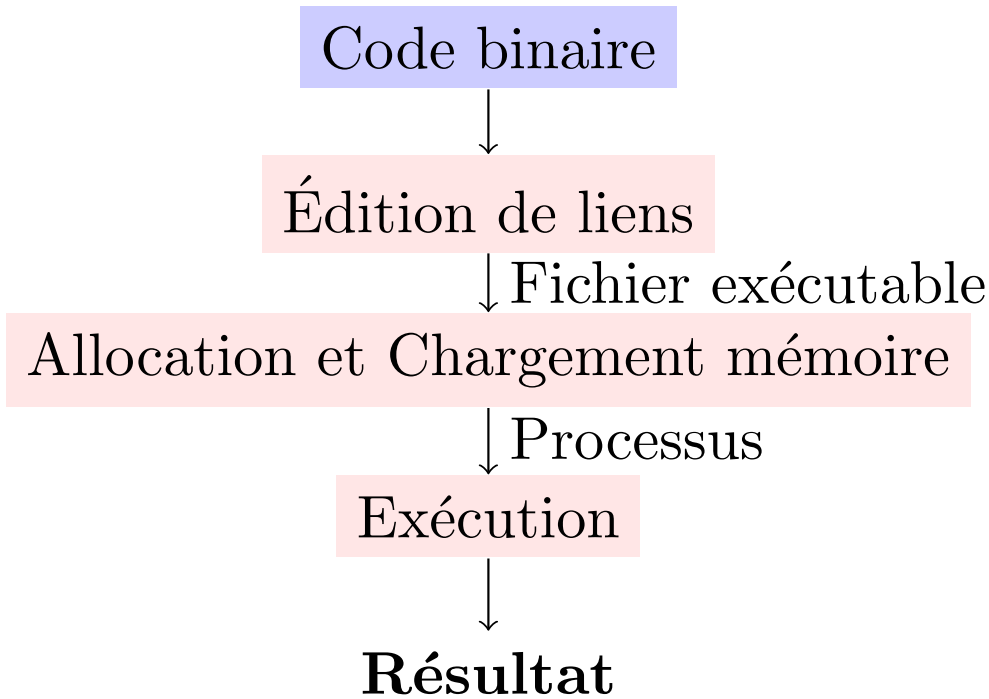
Compilation : Un compilateur est un programme informatique qui transforme un code source écrit dans un langage de programmation (le langage source) en un autre langage informatique (le langage cible). (source *Wikipedia*)

Dans ce cours : Nous allons prendre un langage source celui d'une calculette scientifique (incluant la définition de fonction) et pour langage cible celui de la MVàP (*Machine Virtuelle à Pile*, un langage bas-niveau sur lequel nous reviendrons plus en détail dans la suite.

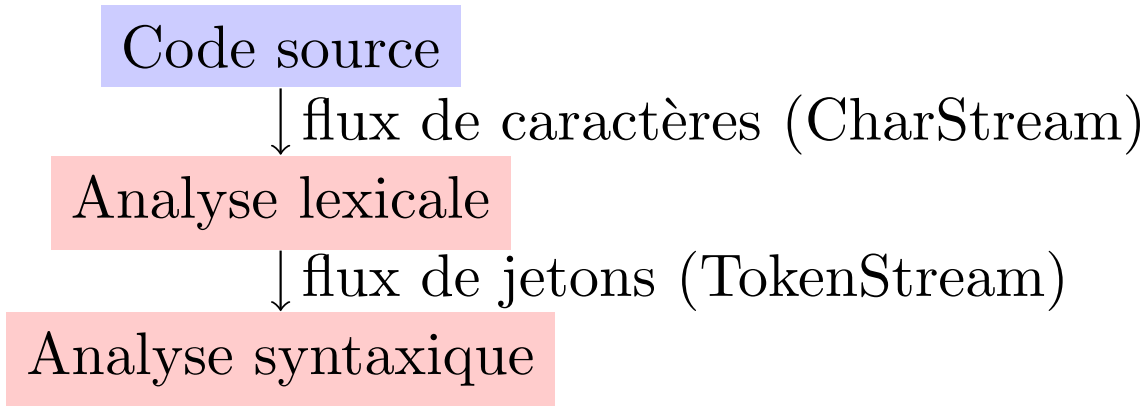
Les grandes étapes de la compilation



... jusqu'à l'exécution



2. L'analyse Lexicale


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

Exemple d'analyseur lexical en Antlr

grammaire (fichier Ex_lex.g4)

```
1  lexer grammar Ex_lex;

3  // Lexer
   NUMBER: ('0'..'9')+;
5
   ID: ('a'..'z')('a'..'z'|'0'..'9')+;
7
   OP: ('+'|'-'|'*'|'/');
9
   WHITE_SPACE : (' '|'\n'|'\t'|'\r')+;
11
   UNMATCH:.;
```

flux de texte

```
12␣+␣golgot14
-678
```

flux de jetons

```
[@0,0:1='12',<1>,1:0]
[@1,2:2=' ',<4>,1:2]
[@2,3:3='+',<3>,1:3]
[@3,4:4=' ',<4>,1:4]
[@4,5:12='golgot14',<2>,1:5]
[@5,13:14=' \n',<4>,1:13]
[@6,15:15='-',<3>,2:0]
[@7,16:18='678',<1>,2:1]
[@8,19:19=' ',<4>,2:4]
[@9,20:19='<EOF>',<-1>,2:5]
```

Commande

```
grun Ex_Lex 'tokens' -tokens < entree.Ex_lex
```

Analyse lexicale : vocabulaire et définition

Lexème : chaîne de caractère 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 convertie une 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, celui-ci est à la fois identifié comme un *mot clé* et comme un *identifiant*.
Gestion via des priorité : ici on choisira *mot-clé* comme unité lexicale.

En Antlr

ANTLR resolves lexical ambiguities by matching the input string to the rule specified first in the grammar

```
BEGIN : 'begin' ;
ID : [a-z]+ ;
```

Gestion des ambiguïtés

1 lexème est le préfixe d'1 autre.
Il y a 2 approches duales.

L'approche gloutonne (greedy). Pour un identifiant, ou un entier, on conserve la plus longue correspondance.

En Antlr
C'est le comportement par défaut pour le *lexer*. Par exemple, `beginner` sera reconnu comme ID et non pas comme `begin` suivi de l'ID `ner`.

Gestion des ambiguïtés

L'approche non gloutonne (not greedy).
On prend la plus petite correspondance.
Typiquement utilisée pour une chaîne de caractère ou un commentaire,

En Antlr

```
STRING : '"' .* '"' ;
```

La règle ci-dessus détecterait un seul jeton pour `"mot1""mot2"` On dispose de `?` pour indiquer qu'on veut le mode non glouton.

```
1 STRING : '"' .*? '"' ;
```

Notons au passage que si on veut pouvoir utiliser " à l'intérieur d'une chaîne de caractère en la protégeant avec un \ (comme en python) il faut échapper ce caractère aussi en Antlr (également avec \).

```
1 STRING: '"' (ESC|.)*? '"' ;
  fragment
3 ESC : '\\"' | '\\\\';
```

Analyseur lexical en Antlr

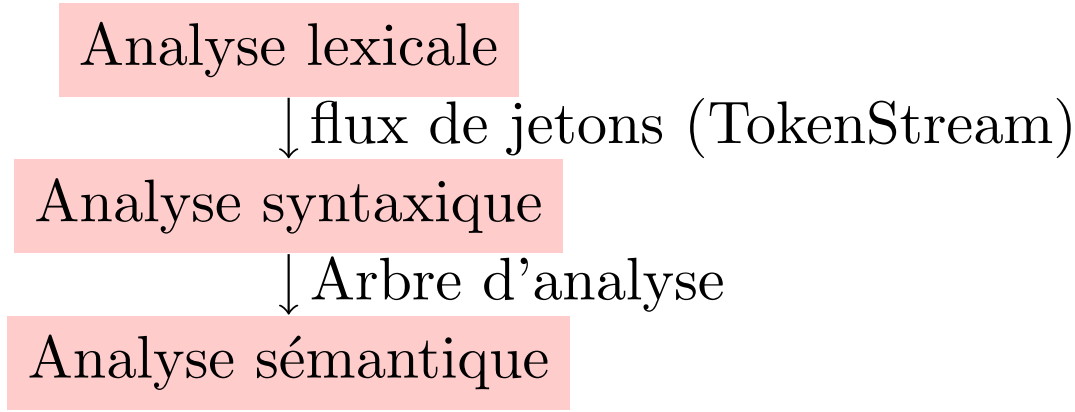
Pas de séparation : les règles pour l'analyseur lexical et l'analyseur syntaxique sont généralement mises dans le même fichier.

Convention : les atomes correspondant a des règles lexicales sont regroupées et mis en majuscules.

Sous le capot, création d'un analyseur lexical en faisant.

```
1 import org.antlr.v4.runtime.*;
  // <snip>
3 Ex_lexLexer  lex = new Ex_lexLexer(new ANTLRFileStream(
      args[0]));
  CommonTokenStream tokens = new CommonTokenStream(lex);
```

3. Analyse syntaxique


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

Exemple d'analyseur syntaxique en Antlr

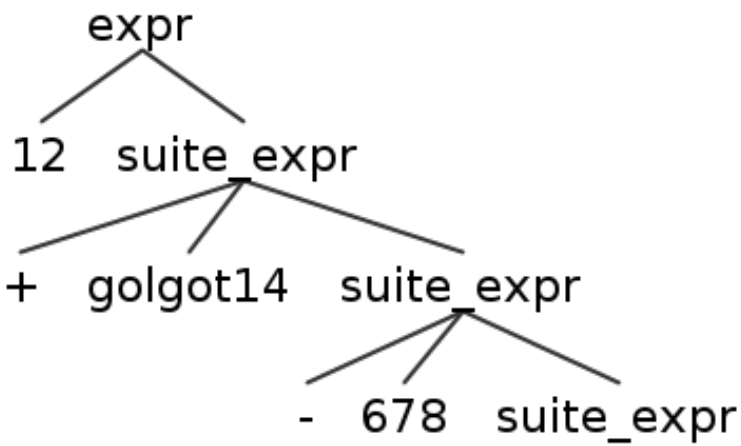
grammaire (fichier `Ex_lex.g4`)

```
grammar Ex_pars;

// Parser
expr
    : (NUMBER|ID) suite_expr
    ;

suite_expr
    : OP (NUMBER|ID) suite_expr
    | /* vide */
    ;

// Lexer
...
```

flux de jetons

```
[@0,0:1='12',<1>,1:0]
...
[@9,20:19='<EOF>',<-1>,2:5]
```

arbre syntaxique



Analyse syntaxique

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 qu'elle 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 appliquer.

Choix :

- gérer les ambiguïtés : s'il existe deux arbres de dérivations correspondant à une expression, on utilise des informations de priorité ou de associativité.
- Gérer l'incorrect : si l'a grammaire n'est pas `LL*`, il existe un certain nombre de mécanisme.

Analyse syntaxique en Antlr

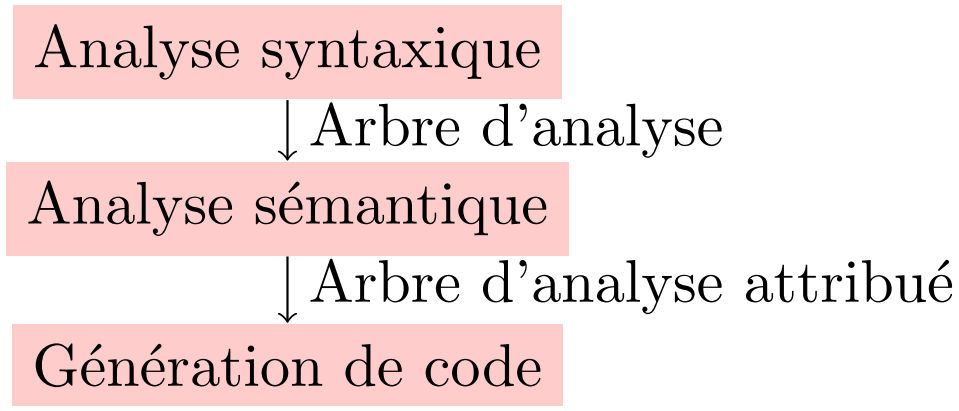
Objet de base : on manipulera souvent directement l'arbre d'analyse et on évitera de la transformer en AST.

Convention : les non-terminaux seront mis en minuscule.

Sous le capot, création d'un analyseur syntaxique en faisant.

```
//snip
Ex_parsParser parser = new Ex_parsParser(tokens);
parser.calcul();
```

4. Analyse sémantique


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

Exemple en Antlr

```
1 // <...>
  suite_expr returns [int val]
3     : OP NUMBER s=suite_expr { $val = 1 + $s.val;}
      | /* vide */ { $val = 0;}
5     ;
  // <...>
```

Analyse sémantique

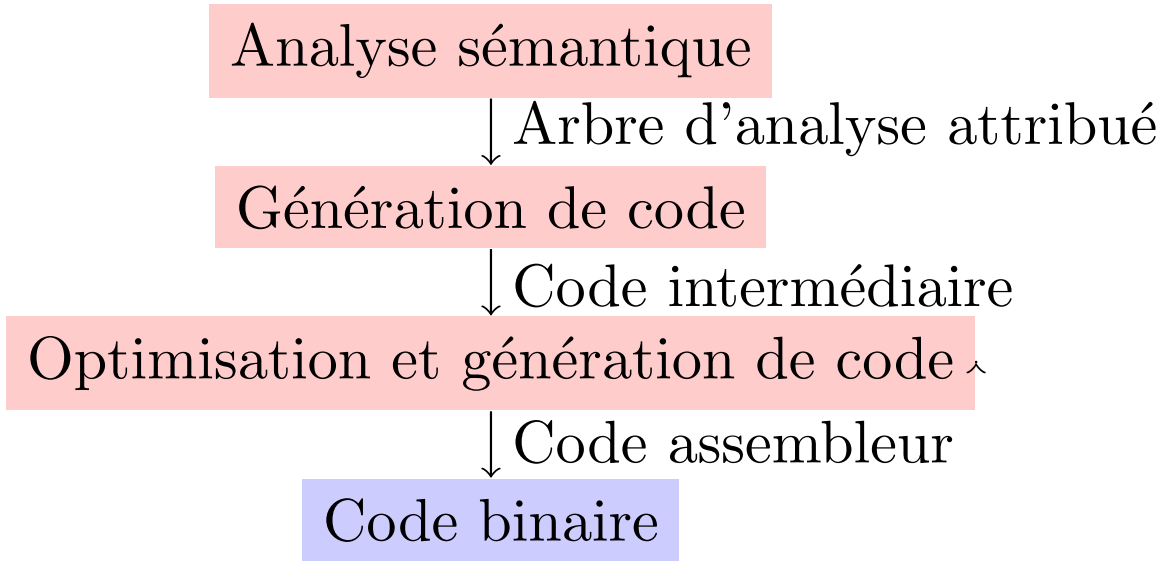
Objectif : Donner un sens à l'arbre de dérivation.

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

Informations :

- Vérification des types ;
- Résolution des noms construction de la table des symboles ;
- Affectation.
-

5. Génération de Code


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

Génération de code

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.

Rappel sur le fonctionnement d'un ordinateur

- Les registres, le bus, la mémoire ;
- Les registres PC, SP ;
- ...

Code à trois adresses (three-address code TAC)

Instructions typiques :

Chargement d'un registre à partir d'une adresse mémoire	$R1 \leftarrow a$	`load R1 a`
Stockage en mémoire du contenu d'un registre	$b \leftarrow R2$	`store b R2`
Opérations binaires, par ex. addition	$R3 \leftarrow R1 + R2$	`add R3 R1 R2`
Branchement		`goto L`
Branchement conditionnel	if $R1 == 0$ then goto L	

Génération de code

Mise en place sur un exemple :
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).

Exemple de code pour les expressions simples

Exemple : $x \leftarrow (a - b) * (c + d)$

Exemple de code pour les expressions simples

Exemple : $x \leftarrow (a - b) * (c + d)$

R1 ← a	load R1 a
R2 ← b	load R2 b
R3 ← a - b	sub R3 R1 R2
R4 ← c	load R4 c
R5 ← d	load R5 d
R6 ← c + d	add R6 R4 R5
R7 ← R3 * R6	mult R7 R3 R6
x ← R7	store x R7

Code à pile

Instructions typiques :

Ajouter sur la pile	`PUSH a`
Addition	`ADD`
Stockage	`STORE x`

Avantages :

- Forme compacte ;
- Pas de registre à nommer ;
- Simple à produire, simple à exécuter.

Désavantages :

- Les processeurs opèrent sur des registres, pas des piles ;
- Il est difficile de réutiliser les valeurs stockées dans la pile .

Exemple de code pour les expressions simples

Exemple : $x \leftarrow (a - b) * (c + d)$

Exemple de code pour les expressions simples

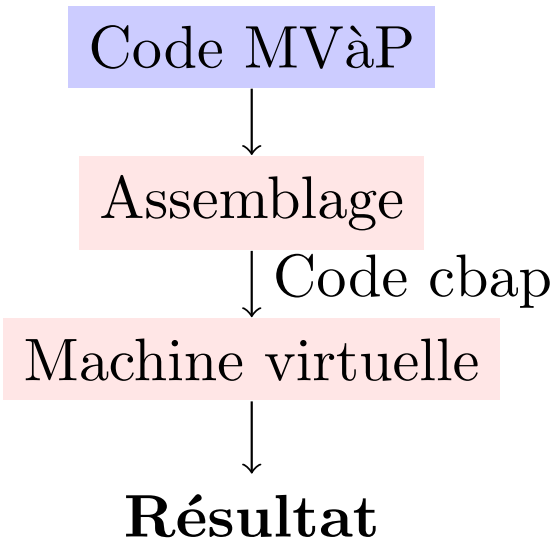
Exemple : $x \leftarrow (a - b) * (c + d)$

PUSH a
PUSH b
SUB
PUSH c
PUSH d
ADD
MULT
STORE x

Deuxième partie II

La Machine Virtuelle à Pile

Principe



Code MVàP

```
PUSHI 5
PUSHI 8
MUL
PUSHI 2
PUSHI 1
MUL
ADD
WRITE
HALT
```

Version assemblée de ce code

```
 Adr |  Instruction
-----+--------------
   0 | PUSHI 5
   2 | PUSHI 8
   4 | MUL
   5 | PUSHI 2
   7 | PUSHI 1
   9 | MUL
  10 | ADD
  11 | WRITE
  12 | HALT
```

Concètement.

```
$ java MVaPAssembler test.mvap
```

produit un fichier `test.mvap.cbap` qu'on peut exécuter en faisant

```
$ java CBaP test.mvap.cbap
   42
```

L'option `-d` permet dans les 2 cas d'avoir plus de détails.

Trace d'exécution

```
$ java CBaP -d test.mvap.cbap
  pc |                    |    fp   pile
=============================================================
   0 | PUSHI        5 |      0 [ ] 0
   2 | PUSHI        8 |      0 [ 5 ] 1
   4 | MUL            |      0 [ 5 8 ] 2
   5 | PUSHI        2 |      0 [ 40 ] 1
   7 | PUSHI        1 |      0 [ 40 2 ] 2
   9 | MUL            |      0 [ 40 2 1 ] 3
  10 | ADD            |      0 [ 40 2 ] 2
  11 | WRITE          |      0 [ 42 ] 1
     42
  12 | HALT           |      0 [ 42 ] 1
```

Fonctionnement

Contenu :

- Quatre registres spéciaux pc , sp, fp, gp ;
- Un segment de code ;
- Une pile.

Actions :

- Toutes les actions modifient les registres ou le contenu de la pile.

Généralités

- La mémoire est organisée en mots, on y accède par une adresse qui est représentée par un entier.
- Les valeurs simples sont stockées dans une unité de la mémoire.
- Une partie de la mémoire est réservée aux instructions du programme
- Un registre stocke l'adresse de l'instruction en cours d'exécution pc (program Counter)
- La taille du code est connue à la compilation
- Le code s'exécute de manière séquentielle sauf instruction explicite de saut
- Un registre stocke l'adresse de la première cellule libre de la pile (sommet de pile) sp (Stack Pointer)
- Un registre stocke l'adresse de la base de la pile gp (Global Pointer)
- Les variables locales sont stockées dans la pile P

PUSH

Code	Pile	sp	pc	Condition
PUSHI n	P[sp] := n	sp+1	pc+1	n est une valeur entière
POP		sp-1	pc+1	sp $>$ 1

- La commande **PUSHI** attend un argument n qui doit être un entier (sinon erreur d'exécution).
- Si cette exécution a lieu alors que la pile vaut P, que le registre de sommet de pile vaut sp et le compteur de programme vaut pc, alors après l'exécution du programme, la pile est modifiée (valeur n à l'adresse sp) ; les registres sp et pc sont incrémentés de 1.

Opérations arithmétiques

Code	Pile	sp	pc	Condition
ADD	P[sp-2] :=P[sp-2] + P[sp-1]	sp-1	pc+1	2 entiers en sommet de pile
SUB	P[sp-2] :=P[sp-2] - P[sp-1]	sp-1	pc+1	
MUL	P[sp-2] :=P[sp-2] * P[sp-1]	sp-1	pc+1	
DIV	P[sp-2] :=P[sp-2] / P[sp-1]	sp-1	pc+1	

Division : La division produit une erreur si le dividende est nul.

Lecture / Affichage

Code	Pile	sp	pc	Condition
READ	P[sp] := entier lu	sp+1	pc+1	un entier sur l'entrée standard
WRITE		sp	pc+1	

Fin de programme

Code	Pile	sp	pc	Condition
HALT				

7. Mémoire

Encodage

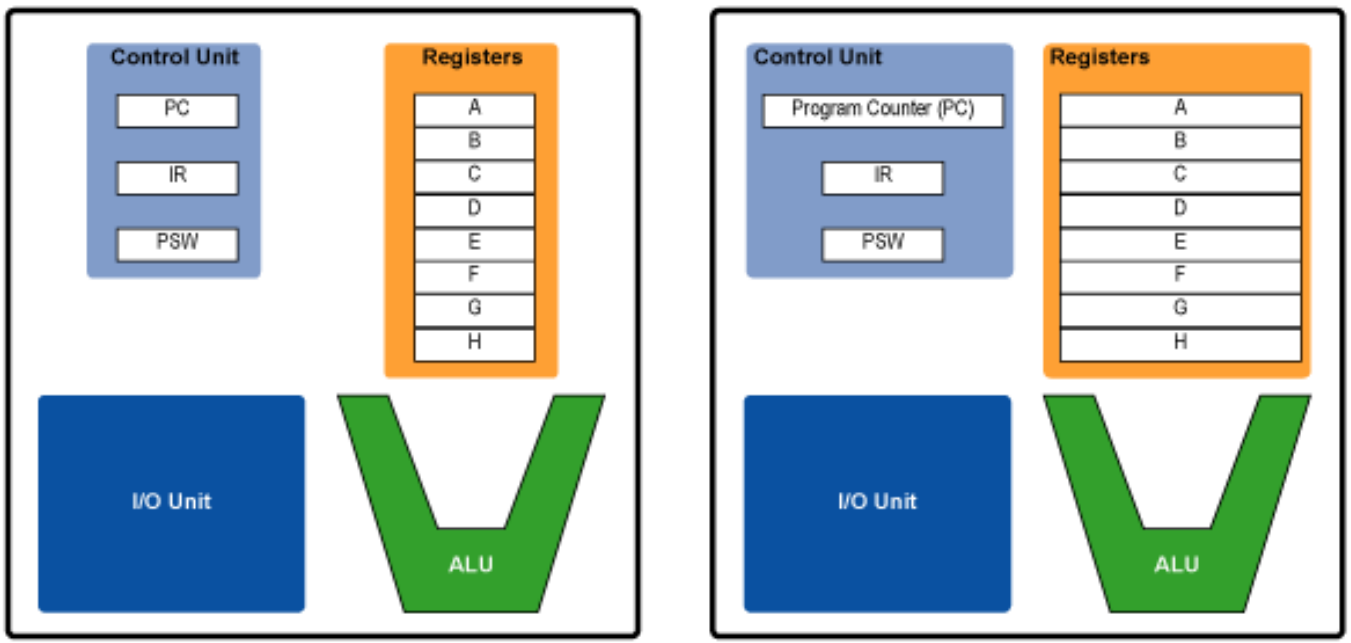
Fait : la mémoire informatique fonctionne sous forme de bits 0 ou 1 (`b`). Ces bits sont groupés par 8 pour former des octets (bytes an anglais, `B`).

Alignement et mots : l'espace élémentaire (appelé mot) varie selon le processeur (32,64,128,... bits). Les données doivent être alignées en mémoire.

Encodage

Fait : la mémoire informatique fonctionne sous forme de bits 0 ou 1 (`b`). Ces bits sont groupés par 8 pour former des octets (bytes an anglais, `B`).

Alignement et mots : l'espace élémentaire (appelé mot) varie selon le processeur (32,64,128,... bits). Les données doivent être alignées en mémoire.



Nombres

Représentation : Il existe trois grandes représentations des nombres en mémoire :

- **Entier non-signé** : l'écriture en binaire standard ;
- **Entier signé** : écriture en complément à 2 ;
- **En virgule flottante** : écriture mantisse / exposant.

Endianess. Dans le cas où l'on manipule un entier de taille plus grande qu'un mot mémoire, on peut le décomposer en plusieurs blocs qui sont rangés :

- poids fort en tête : big endian (ex : Motorola,SPARC) ;
- poids faible en tête : small endian (ex : x86) ;
- bizarrement : middle-endian.

Chaînes de caractères

Encodage. Les chaînes de caractères sont transformées en suite d'entiers à l'aide d'une table de codage de caractère. Il en existe plusieurs :

- **ascii** : l'historique ;
- **latin-1** : pour les langues ouest-européennes (avec latin-9) ;
- **UTF-8** : très général.

Le codage peut se faire en taille fixe ou en taille variable.

Une fois transformée, la chaîne peut être stockée :

- avec un délimiteur de fin : `\0` (ex : C, python, ...) ;
- en indiquant avant la longueur de la chaîne (ex : fortran).

Autres

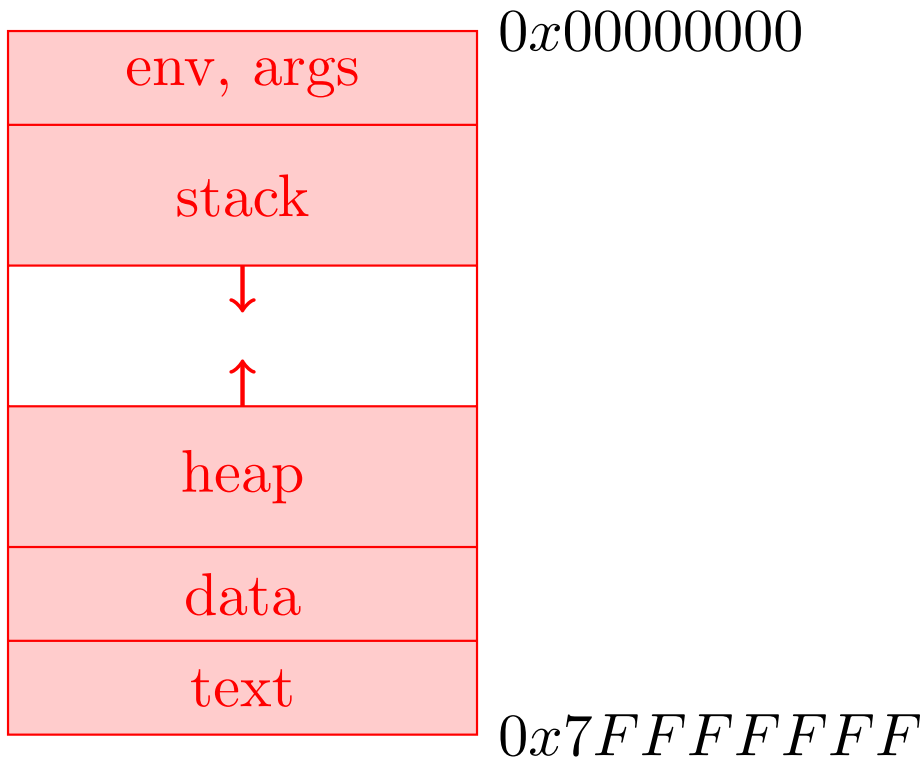
Règles : tous les objets sont manipulés en binaire dans l'ordinateur.

Code Exécutable : le code exécutable entre évidemment dans cette catégorie.

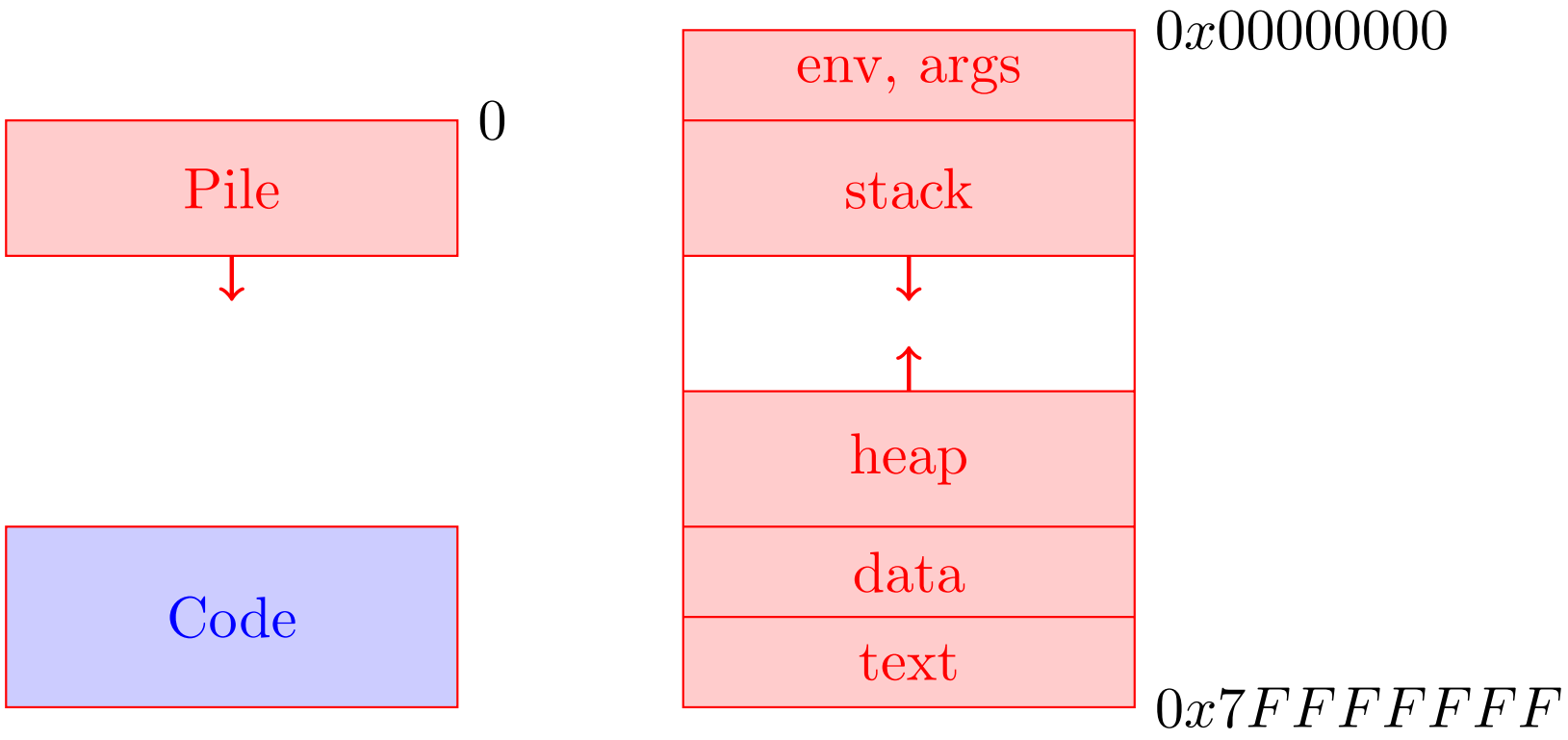
Pointeur : il est possible également de stocker un entier qui indique une adresse en mémoire. On parle alors de pointeur.

Organisation

Vue de l'exécutable : un espace mémoire continu divisé en segments



Cas de la machine à Pile



Cas de la machine à Pile

Spécificités :

- Pas de tas ;
- Une pile contenant déjà les opérations de bases ;
- Une espace dédié et séparé pour le code.

Limitations :

- Très difficile de faire l'allocation de taille inconnue à la compilation.
- Impossible de modifier le code à la volée.

8. Variables (globales)

Portée

Porté d'une variable.

- En général, une variable globale est déclarée en début du fichier source et est valable jusqu'à la fin de l'exécution ;
- Dans certains langages, il est possible de définir des variables à l'intérieur de blocs. Leur portée est limité à ce bloc.

Cas du C

```
#include <stdio.h>

int main (void) {
  int j;
  j = 0;
  for (int i = 0; i <10; i++) {
    j = i * j + 2;
  }
  printf("%d,%d",j,j);
}
```

Mise en place

On peut :

- Empiler la valeur d'une variable globale qui est contenue dans la pile
- Recopier la valeur en sommet de pile dans la variable globale

Mise en place

On peut :

- Empiler la valeur d'une variable globale qui est contenue dans la pile
- Recopier la valeur en sommet de pile dans la variable globale

Accès direct à la mémoire.

Code	Pile	sp	pc	Condition
PUSHG n	P[sp] := P[gp+n]	sp+1	pc+1	n entier t.q. gp+n < sp
STOREG n	P[gp+n] := P[sp-1]	sp-1	pc+1	n entier t.q. gp+n < sp

À propos de *gp* :

Dans toute la suite du cours, on aura toujours $gp = 0$.

Mise en place

On peut :

- Empiler la valeur d'une variable globale qui est contenue dans la pile
- Recopier la valeur en sommet de pile dans la variable globale

Accès direct à la mémoire.

Code	Pile	sp	pc	Condition
PUSHG n	P[sp] := P[gp+n]	sp+1	pc+1	n entier t.q. gp+n < sp
STOREG n	P[gp+n] := P[sp-1]	sp-1	pc+1	n entier t.q. gp+n < sp

À propos de *gp* :

Dans toute la suite du cours, on aura toujours $gp = 0$.

Variables et compilation

Besoin :
Déclaration : réservation de l'espace mémoire et mémorisation du liens entre nom et emplacement mémoire.

Assignation / Utilisation : retrouver l'emplacement mémoire et l'utiliser pour stocker ou charger la valeur.

Structure :
Pour stocker ces informations, on utilise une table des symboles qui contient l'ensemble des symboles présent dans le source et permet en cas de besoin d'associer entre autre une adresse.

En Antlr

Membres : Il est possible de déclarer des structures utilisées sur l'ensemble de l'analyse :

```
@members {
HashMap<String, Integer> memory = new HashMap<String,
    Integer>();
}
```

et d'inclure des headers java :

```
@header {
import java.util.HashMap;
}
```

En Antlr

Membres : Il est possible de déclarer des structures utilisées sur l'ensemble de l'analyse :

```
@members {
HashMap<String, Integer> memory = new HashMap<String,
    Integer>();
}
```

et d'inclure des headers java :

```
@header {
import java.util.HashMap;
}
```

On utilisera cette facilité pour manipuler notre table de symboles.

9. Branchements

Branchement inconditionnel

Changer : pour changer l'exécution, il suffit de changer la valeur du pc.

Code MVàP :

Code	Pile	sp	pc	Condition
JUMP label		sp	instr(label)	

Pour déterminer où aller, on indique des étiquettes dans le code MVàP.

```
1 <...>
  JUMP blop
3 PUSHI 1
  LABEL blop
5 <...>
```

Coût : Dans un processeur réel, une telle opération est coûteuse car elle nécessite de "casser" le pipeline.

Branchement conditionnel

Principe : L'instruction de branchement est exécuté suivant une condition.

Code MVàP :

Code	Pile	sp	pc	Condition
JUMPF label		sp-1	pc+1 si P[sp-1] $\neq 0$ instr(label) sinon	

Conditions dans MVàP

Instructions MVàP :

Code	Pile	sp	pc
INF	P[sp-2] := 1 si P[sp-2] $<$ P[sp-1], 0 sinon	sp-1	pc+1
INFEQ	P[sp-2] := 1 si P[sp-2] $\leq$ P[sp-1], 0 sinon	sp-1	pc+1
SUP	P[sp-2] := 1 si P[sp-2] $>$ P[sp-1], 0 sinon	sp-1	pc+1
SUPEQ	P[sp-2] := 1 si P[sp-2] $\geq$ P[sp-1], 0 sinon	sp-1	pc+1
EQUAL	P[sp-2] := 1 si P[sp-2] $=$ P[sp-1], 0 sinon	sp-1	pc+1
NEQ	P[sp-2] := 0 si P[sp-2] $=$ P[sp-1], 1 sinon	sp-1	pc+1

Retour vers la machine à pile ...

Exemple de code :

```
1  <...>
   PUSHI 12
3  PUSHI 24
   INFEQ
5  JUMPF suite
   PUSHI 13
7  JUMP fin
   LABEL suite
9  PUSHI 2
   LABEL fin
11 PUSHI 3
   ADD
13 WRITE
   <...>
```

... et le C

```
<...>
LBB0_1:                                 ## =>This Inner
      Loop Header: Depth=1
          cmpl    $20, -8(%rbp)
          jge     LBB0_4
## BB#2:                                ##    in Loop:
     Header=BB0_1 Depth=1
          movl    -12(%rbp), %eax
          imull   $3, -8(%rbp), %ecx
          addl    %ecx, %eax
          addl    $5, %eax
          movl    %eax, -12(%rbp)
## BB#3:                                ##    in Loop:
     Header=BB0_1 Depth=1
          movl    -8(%rbp), %eax
          addl    $1, %eax
          movl    %eax, -8(%rbp)
          jmp     LBB0_1
LBB0_4:
<...>
```

Boucles

Structures classiques :

```
1 <...>
    | 'if' '(' c=condition ')' blocthen=instruction ('
        else' blocelse=instruction)?
3   | 'while' '(' c=condition ')' i=instruction
    | 'for' '(' init=assignation? ';' c=condition? ';'
        incr=assignation? ')' i=instruction
5   | 'do' i=instruction 'until' '(' c=condition? ')'
      finInstruction
```

Compilation :

- Évaluer les conditions ;
- Mettre les étiquettes ;
- Ajouter les sauts conditionnels ;
- Compiler le code.

Exemple avec la machine à pile

```
var i
i = 6
while (i < 10) i =
     i + 1
i
```

```
  PUSHI 0
  JUMP 0
LABEL 0
  PUSHI 6
  STOREG 0
LABEL 1
  PUSHG 0
  PUSHI 10
  INF
  JUMPF 2
  PUSHG 0
  PUSHI 1
  ADD
  STOREG 0
  JUMP 1
LABEL 2
  PUSHG 0
  WRITE
  POP
  HALT
```

```
 Adr |   Instruction
-----+----------------
   0 | PUSHI 0
   2 | JUMP  4
   4 | PUSHI 6
   6 | STOREG 0
   8 | PUSHG 0
  10 | PUSHI 10
  12 | INF
  13 | JUMPF 24
  15 | PUSHG 0
  17 | PUSHI 1
  19 | ADD
  20 | STOREG 0
  22 | JUMP  8
  24 | PUSHG 0
  26 | WRITE
  27 | POP
  28 | HALT
```

La suite

- À 10h15 TD en salle TP (découverte de la MVàP).
- jeudi, début d'une séquence de TP en vu de soumettre votre DM.